User-friendly Formal Methods for Security-aware Applications and Protocols

POLITECNICO DI TORINO
SCUOLA DI DOTTORATO
Dottorato in Ingegneria Informatica e dei Sistemi – XXVII ciclo

Tesi di Dottorato

User-friendly Formal Methods for Security-aware Applications and Protocols

Piergiuseppe Bettassa Copet

Tutore
prof. Riccardo Sisto

Coordinatore del corso di dottorato
prof. Matteo Sonza Reorda

2016

Abstract

Formal support in the design and implementation of security-aware applications increases the assurance in the final artifact. Formal methods techniques work by setting a model that unambiguously defines attacker capabilities, protocol parties behavior, and expected security properties. Rigorous reasoning can be done on the model about the interaction of the external attacker with the protocol parties, assessing whether the security properties hold or not.

Unfortunately, formal verification requires a high level of expertise to be used properly and, in complex systems, the model analysis requires an amount of resources (memory and time) that are not available with current technologies.

The aim of this thesis is to propose new interfaces and methodologies that facilitate the usage of formal verification techniques applied to security-aware protocols and distributed applications. In particular, this thesis presents: (i) Spi2JavaGUI, a framework for the model driven development of security protocols, that combines (for the first time in literature) an intuitive user interface, automated formal verification and code generation; (ii) a new methodology that enables the model driven development and the automated formal analysis of distributed applications, which requires less resources and formal verification knowledge to complete the verification process, when compared to previous approaches; (iii) the formal verification of handover procedures defined by the Long Term Evolution (LTE) standard for mobile communication networks, including the results and all the translation rules from specification documents to formal models, that facilitates the application of formal verification to other parts of the standard in the future.

Summary

Nowadays, distributed network applications are becoming even more pervasive. However, data security of is at risk if protection mechanism are flawed or obsolete. In fact, malicious attacker can exploit new technologies to break security mechanism that were considered adequate years ago.

Throughout the years, formal verification approaches have been developed and improved to early identify and correct problems in software applications. Formal verification techniques are based on mathematical models and, through logical reasoning, it is possible to verify if the models satisfy the desired security properties. Choosing the most adequate technique, and defining a correct model of the system under analysis are crucial points of the formal verification process. Many tools support the logical reasoning process through the proof. They can be either fully automated, or semi-automated (thus requiring the intervention of the developer). Another possible characterization of these tools is as follows: those that derive a formal model from the final implementation, and those that are based on the model driven design paradigm [1] (in this way, it is possible to automatically generate the code that implements the behaviour described in the formal model, and guarantee that the code has the same properties verified in the formal model).

Goal of this thesis is to propose new methodologies that can facilitate the use of formal verification techniques, especially when applied to distributed applications. These new approaches lower the adoption barriers for the developers (in particular for those wo are not formal verification experts) and, in addition, reduce the resources necessary to complete the verification, if compared to previous solutions. As a consequence, formal verification can be applied to a wider range of software application than before.

The first work presented in this thesis is Spi2JavaGUI (part of the text has been taken from a previous paper [2]): an innovative user-friendly approach to modeling (using a graphical interface), formal verification and automatic generation of code that implements cryptographic protocols following the model driven paradigm.

The second work enables the verification of application dependent security properties, in addition to "classic" security properties (i.e. secrecy and integrity of data, authentication of parties) during the development of distributed applications.

Application-specific properties are closely related to the application under examination. For example, the value of a variable must always be within a specific range, during all possible execution paths that can be followed (including different scheduling possibilities) by the application. Moreover, the entire workflow is completely automated, and is implemented as a major extension of the JavaSPI [3, 4, 5, 6] framework. This section contains text from a previous paper [5].

The last work described in this thesis regards the formal analysis of handover procedures defined in the Long Term Evolution (LTE) standard for mobile communication networks. These procedures are activated when the management of a mobile device connection to the network is handed over from a cell to another cell in the network. The analysis deals with security aspects of the handover procedures that have not been considered in previous works available in literature. A detailed description of the methodology used to translate the specifications of the standards to formal models is given, along with how the issues have been addressed. Part of the text was published in a previous paper [7].

Acknowledgements

Firstly, I would like to express my sincere gratitude to my advisor Prof. Riccardo Sisto for the continuous support of my Ph.D study and related research.

My sincere thanks also goes to Madalina Baltatu, Luciana Costa, Roberta D'Amico (Telecom Italia) and Guido Marchetto (Politecnico di Torino) for their support in the "Formal verification of LTE and UMTS handover procedures" project.

Last but not the least, I would like to thank my family for supporting me spiritually throughout writing this thesis and my my life in general.

Contents

Chapter 1

Introduction

In the last years, distributed applications have become more and more important in communication networks and, in many occasions, the data exchanged must be strongly protected from external malicious agents, which constantly increase their powerfulness thanks to the availability of new technologies and hardware resources.

A distributed application consists of two or more software processes that are executed, at the same time, on separate machines connected by a communication network (e.g. a local network, Internet). In general, communication networks are insecure networks, where messages can be intercepted, arbitrarily modified or deleted by malicious attackers who gain access to the networks, with the aim to exploit vulnerabilities in one or more processes in order to alter the behaviour of the application, and obtain or alter private and sensible data.

Therefore, it is necessary to adopt protection and data authenticity verification systems. The scientific literature describes numerous examples of vulnerabilities discovered in applications and protocols, even years after they were introduced and used [8, 9, 10, 11, 12, 13]. Over the years many solutions have been proposed to identify problems in distributed applications and ensure the absence of errors.

However, designing and implementing a distributed application still requires a high level of expertise in the various fields involved in the development application workflow.

Vulnerabilities, in fact, may be introduced in each of the different application development steps. For example, there may be errors in the initial requirements definition phase (which may also be very complex), or during their translation in the code that implements the application. Moreover, vulnerabilities may already be present in the external libraries that are reused. Finally, the extreme optimization that are introduced in the code on order to obtain better performance may create further problems: generally, an optimized implementation is difficult to be analyzed because it minimizes operations, often following a complex logic.

To identify and correct problems in software applications, over the years, formal verification techniques have been developed. In general, formal verification can

be applied in various fields, such as the verification of hardware devices, or mixed software/hardware systems. Formal verification techniques are based on mathematical models, and can clearly define, through logical reasoning supported by automated tools or semi-automatic, if the mathematical model satisfies certain properties that must be specified to the model. Unfortunately, in almost all the cases, formal verification cannot be performed directly on the final code that implements the application, because it would require an amount of resources (memory and time) that are not available with current technologies. Therefore, it is necessary to define models of the applications that have to be analyzed. Formal models must be proper abstractions of the applications, without being too complex (because they would be unfeasible to analyze) and, at the same time, not too simple (in order to obtain significant results). Defining a correct model is a key point of the formal verification process. Several techniques have emerged over the years in the field of formal verification, each with different characteristics regarding the analysis of cryptographic functions (e.g. computational or symbolic), and the behaviour of the malicious agents. The scientific community has proposed different methodologies, tools and frameworks reduce the complexity of formal verification processes, and enable inexperienced developers to use it, by automating some phases or all of the verification process. These tools fall into two main categories: those that are able to derive a formal model starting from the final implementation, and those that are based on the model driven design paradigm [1]. The latter are able to generate the code that implements the specifications of the formal model, and ensure that the generated code respects the same properties verified in the initial model. Both techniques are successfully used [14]: the choice of the method to use depends on various conditions, for example, the model-derivation is used mainly when it is necessary to analyze an already existing implementation. In this way it is possible to obtain a formal model considering the entire application. On the contrary, the model driven development way is is mainly used in when there is no implementation, and it is convenient to obtain an automatically generated implementation starting from a formal model, on which all the desired properties have been verified. However, in most cases, the model driven generation processes is not able to produce a complete and running implementation in all its aspects, because the initial model, having to be an abstraction of the final implementation, cannot contain all the details that are required in the final code. For this reason it is necessary to manually add or edit the generated implementation. Therefore, the generation process do not generate a highly optimized code, but the result is a compromise between efficiency and usability, in order to allow the necessary modifications without altering the validity of the properties verified in the formal model.

1.1 Contribution

Part I of this thesis presents new methodologies to facilitate the use of formal verification techniques in the verification of distributed applications, which lower the adoption barriers for the developers that are not formal verification experts and, moreover, reduce the resources necessary to complete the verification than previous existing solutions.

Chapter 3 describes Spi2JavaGUI, which is a user-friendly approach to modeling, formal verification and implementation of cryptographic protocols following the model driven paradigm (part of the text has been taken from a previous paper [2]). Cryptographic protocols are a class of communication protocols, that use cryptographic functions to secure messages exchanged among processes that compose the distributed applications, in order to provide data privacy and/or authentication of parties (i.e. processes) that are involved in the protocol. In this context, Spi2JavaGUI is a framework that combines a graphical editor (implemented as a plugin for the Eclipse platform), formal verification (with symbolic model), and the generation of Java code that implements the protocol. The code enforces the security properties verified in the formal model.

Spi2JavaGUI generates a formal model that can be analyzed by ProVerif [15, 16], a tool for the automatic verification of cryptographic protocol models. The ProVerif models are based on the Dolev-Yao abstraction [17], which uses a symbolic representation for data and cryptographic operations, and supposes that the malicious attacker has complete control over public communication channels (i.e. the attacker can read data in transit, forge and send new messages, drop or alter messages that are exchanged). More details about ProVerif are given in Section 2.1.

Chapter 4 presents a new methodology for the formal analysis and the model driven development of distributed applications (part of the text was published in a previous paper [5]). This procedure exploits JavaSPI [3, 4, 5, 6], a framework similar to Spi2JavaGUI, but where the graphical interface is replaced with a Java-like model. Thus, the JavaSPI framework is easy to use for those who already know the Java language. Moreover, thanks to the libraries included in the framework, it is possible to simulate (according to the Dolev-Yao abstraction [17]) and debug the JavaSPI model as a standard Java application, in order to early identify errors and problems. Similarly to Spi2JavaGUI, Spi2Java can generate a ProVerif model in order to perform formal verification. JavaSPI was previously developed in the research group of Prof. Sisto, and further extended in this work.

The methodology proposed here enables the verification of application dependent security properties, in addition to "classic" security properties (i.e. secrecy and integrity of data, authentication of parties). An application-specific property depends heavily on the application under examination. For example, the value of a variable must always be greater than zero for all possible execution paths that can be

followed when the different processes (that compose the distributed application) are running. The new methodology uses a modified version of JavaSPI to define the communication protocol between the different actors of the application. The JavaSPI model contains the authentication and secrecy properties that must be verified, using the automatically generated ProVerif model.

After completing the verification with the ProVerif tool, it is necessary to generate a set of Java classes that implements the protocol with all the details (e.g. algorithms used, length of keys). The final distributed application must use only the generated code for the communications between the various processes that constitutes the application.

The second phase of the analysis verifies the application specific security properties. The Java Pathfinder (JPF) [18] tool is used in this stage. JPF is a model-checker for Java applications, and verifies executable Java bytecode in order to check if specific properties (defined by *listeners*) are satisfied. The simplest approach is to verify the entire Java code of the application (composed of generated protocol code and the remainder of the application) directly with JPF. However, this approach is almost unfeasible because of the combinatorial explosion of the number of states that the analysis must deal with. To limit this problem and enable the verification of the entire application, JavaSPI generates a set of Java classes, called "stub", which implement a "simplified" version of the communication protocol, by omitting details that are not relevant for the model checking analysis. The stub classes must be used to replace the protocol code (previously generated with JavaSPI) in the distributed application, before launching the JPF analysis. The replacement requires changing a few instructions of the application, that can be made easily. Another modification required in the application is to convert all the different processes that compose the application into threads that are created in the same process, because JPF can analyze concurrent threads, not distinct processes. After completing the changes, the JPF analysis can be started to verify all the application-specific security properties. It is worth noting that the complexity of the model checking analysis process (i.e. the time and memory required) depends on the complexity of the application and the properties to verify. Moreover, the analysis with a model checker can not deal with an an infinite number of application sessions (i.e. threads that constitute the application), however, that in the great majority of cases, a few session are sufficient to identify violations of security requirements.

Part II of the thesis presents the formal analysis of procedures defined in the Long Term Evolution (LTE) standard for mobile communication networks. The procedures that have been analyzed were not analyzed previously, or not analyzed at all, in literature, with the security requirements that have been considered in this analysis (part of the text was published in a previous paper [7]). The formal analysis has been performed with the ProVerif tool. In this work the formal analysis is not related to implementation of software applications, but the formal model described

here represent abstractions of the entire communication system, which includes both software and hardware components. The thesis gives a detailed description of the methodology used to translate the specifications of the standards to ProVerif formal models, and how the incompleteness issues have been resolved.

Finally, Part III concludes and resumes the results.

Part I

Formal Methods in model-driven development of secure software

Chapter 2

Background

2.1 ProVerif

ProVerif [15, 16] is a tool for automatic verification of cryptographic protocols, using theorem-proving techniques, where the protocol actors and the attacker are modeled according to the symbolic approach defined by Dolev and Yao [17]. In this model, the attacker has complete control over communications channels and can read, delete, modify messages in transit or forge new messages. The symbolic representation of data and cryptography implies that encryption is considered ideal: the attacker can decipher an encrypted message only when he knows the right key.

As the possible behaviors of the attacker are already pre-defined by the Dolev-Yao approach, when using ProVerif it is enough to model the trusted actors of the protocol, while the attacker model is already available inside ProVerif. An important feature of ProVerif is its ability to model and analyze an unlimited number of sessions of the protocol, even running in parallel, differently from model checkers, which can only analyze a bounded system.

Because of the inherent undecidability of the formal verification problem, ProVerif may report false attacks, i.e. attacks which in reality are not possible. As a consequence, when an attack is reported by ProVerif, in the form of an execution trace that violates the specified property, it is necessary to carefully analyze it in order to understand if it is a real attack. However, if a property is reported as satisfied, then it is guaranteed to be true (ProVerif builds a formal proof for it), and no attack is feasible in the model.

2.2 Spi2Java

Spi2Java [19, 20] is a model driven development framework for the automatic generation of code that implements security protocols. Spi2Java uses a spi calculus

model [21] to define an abstraction of the protocol, and can automatically generate a ProVerif model from the spi calculus model.

The formal language underlying the Spi2Java approach is the domain-specific spi calculus [21] language. A model of a security protocol in spi calculus is composed of a system of parallel processes. Each protocol principal is modeled by a process, exchanging messages with other processes via possibly shared (i.e. insecure) communication channels. Typically, a principal is described as a sequential program that performs a sequence of input/output operations, checks on received data, and cryptographic operations.

The model is Dolev-Yao [17]: messages are represented symbolically as terms of an algebra and cryptographic functions are algebraic operators on these terms. Such operators have the properties that the corresponding functions should ideally fulfill. For example, since $H(x)$ represents the result of applying a cryptographic hash function to x, $H(a)$ and $H(b)$ are always different for different a and b, and there is no operator that takes $H(x)$ and returns x. Conversely, the encryption of message M with key k, represented by term $\{M\}_k$, can be decrypted to get M only if key k is available. The attacker can eavesdrop, alter, drop or forge messages on public communication channels, based on the messages the attacker knows or learned. Finally, terms are untyped, in order to be able to reason about possible attacks based on type confusion.

A spi calculus model is abstract but rigorous. It can be automatically analyzed in order to formally verify that there are no possible attacks on the protocol logic, under the Dolev-Yao assumptions. In Spi2Java, the ProVerif tool is leveraged to perform formal verification. ProVerif is a fully automatic verification engine which accepts (an extension of) spi calculus as its input modeling language.

Semi-automatic code generation from an abstract model expressed in spi calculus can be done using the Spi2Java [19, 22] framework in two steps. First i) the abstract model is refined, by adding low level details (e.g. which cryptographic algorithm must be used for a hash operation, or how to transform a message into its network binary representation). These details are written in a separate document, called the refinement document (in XML format), so as to keep the spi calculus model as simple (and readable) as possible; then ii) the consistency between the refinement document and the spi calculus model is checked and the automated code generator of Spi2Java is invoked to generate the implementation code from the spi calculus model and the refinement document.

The workflow of the Spi2Java framework is depicted in Figure 2.1. In conclusion, Spi2Java can generate interoperable implementations of security protocols, in Java, and that implementations are proven to be without vulnerabilities. A soundness theorem [23, 20] demonstrates the validity of the soundness between the abstract model and the generated implementation.

However, the main drawbacks of the procedure implemented by Spi2Java are:

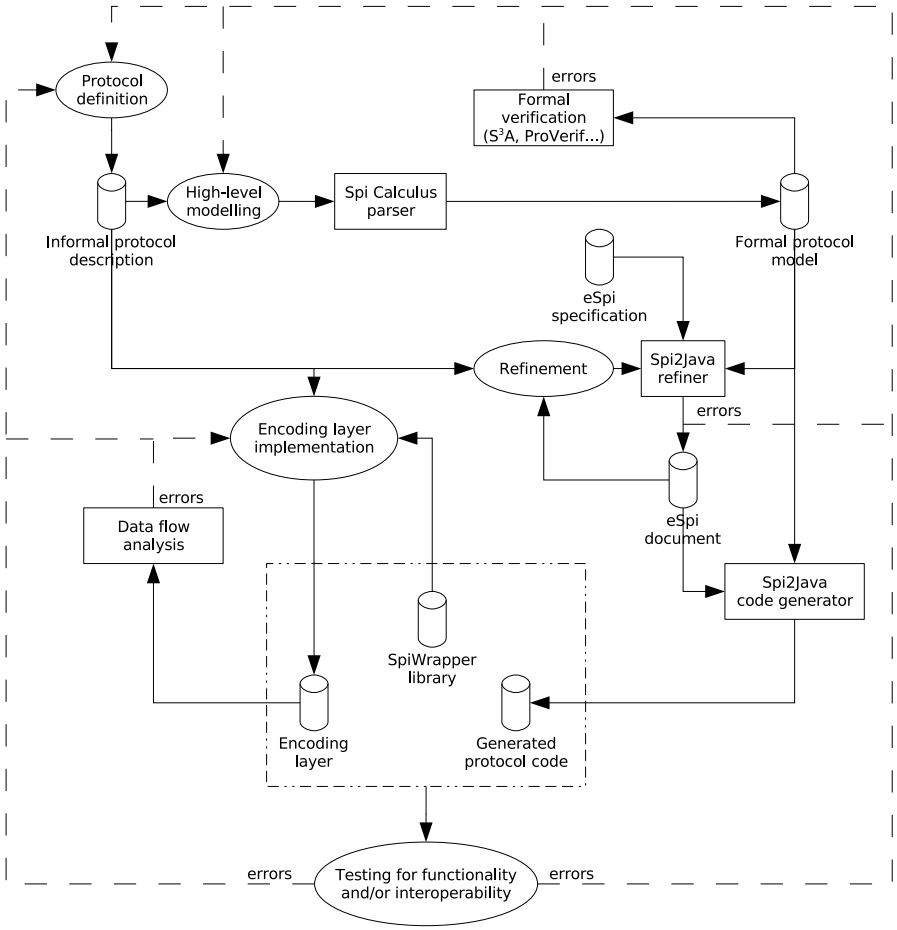

Figure 2.1. Spi2Java workflow

- Spi2Java describes the behaviour of the protocol using spi calculus language, that in some cases is rather complex, especially for large protocols;
- all commands must be called from command-line, and the user is responsible of following the correct workflow;
- the implementation of a protocol takes place in several stages in which refinement actions must be performed on the abstract model (in order to get a concrete and interoperable Java implementation), this involves some complex and time consuming operations from the user to trace logically the terms defined in the model, and that appear in the XML file that contains type specifications. Here too, the complexity increases with the complexity of the protocol that is being modeled;
- subsequent changes to the model require the repetition of the generation

procedure: this means that the intermediate files are regenerated, and the user has to apply again his customizations.

The workflow of Spi2Java exposes the user to the need of knowing the abstract modeling language and to a pair of other relevant problems. The first one is that the user needs to know how the various tools work and interact with each other. The second problem is that model and implementation details are stored separately, with loose cross references. This procedure can be very time-consuming in complex protocols and can introduce errors. In fact, the matching between the spi calculus model's terms and low-level specification document's term is done with a numerical index. So, a change (e.g. adding a term) in spi calculus model invalidates all cross references to low-level specification document, and all previous customizations (i.e. implementation details) are lost. User must redo all changes, considering that indexes of terms are changed, and there is no automatized procedure that can efficiently handle this task. In conclusion, modeling a protocol using spi calculus and Spi2Java requires a high level of expertise, a specialized knowledge and training to effectively apply them. Unfortunately, this makes this type of approach less cost-effective than other methods as security and software engineers cannot easily learn and apply it.

2.3 The JavaSPI framework

JavaSPI [3] is a framework for modeling, formally verifying and implementing cryptographic protocols, according to the paradigm of model-driven development. Initially, the user defines an abstract formal model of the protocol according to the Dolev-Yao modeling approach. This model, being abstract, does not include implementation details such as, for example, hash algorithms and length of cryptographic keys. This model, after proper translation, can be formally verified by ProVerif in order to check that it satisfies some security properties. These properties are generally expressed either as secrecy requirements (the attacker must not be able to know some data) or as correspondence requirements referred to events specified in the abstract model. The latter requirements can be used to express authentication or data integrity properties; for example an authentication requirement could be expressed as $terminate(A, B) \Rightarrow start(B, A)$, which means that each time actor A terminates a session of the protocol apparently with B (i.e. event $terminate(A, B)$ occurs), B has previously started a session of the protocol with A (i.e. event $start(B, A)$ has occurred).

When the user is satisfied with the model and confident about its logical correctness, the missing implementation details can be specified and a Java implementation of the protocol can be automatically generated. JavaSPI is very similar to Spi2Java [19], the main difference being the modeling language: while with Spi2Java a protocol is modeled directly in the formal specification language spi-calculus, JavaSPI lets the

user develop the protocol model in the form of a Java application, written with some restrictions on the Java language and making use of a custom library (JavaSpiSim), which offers the same expressiveness as the spi calculus language. In fact, a formal specification of the protocol compatible with ProVerif can be generated automatically from the Java code. Using Java as the modeling language facilitates users who are familiar with object oriented programming and Java. Moreover, this approach lets the user simulate the execution logic of the protocol by means of a normal Java debugger.

Figure 2.2 shows an excerpt of an abstract model written with JavaSPI. Each model is composed of a number of processes, each one specified by a Java class that extends the `spiProcess` library class. The behavior of a process is specified by defining the `doRun` method, which takes as arguments objects belonging to classes of the `JavaSpiSim` library. These classes represent the data types admitted in a security protocol model and include methods for performing common operations, such as for example encrypting or decrypting data or sending or receiving data on channels. The occurrence of an event is specified by calling the `event` method which can have any number of arguments (e.g. `event("start",A,B)` generates event $start(A, B)$).

The implementation details that are necessary for generating the final implementation code are specified as Java annotations added to the abstract model. JavaSPI shares with Spi2Java the same code generation mechanism, which has been proved to preserve a large class of security properties [23]. This means that if a security property has been proved to hold on the formal model, then that property holds on the automatically generated Java implementation too.

2.4 Java Pathfinder

Java Pathfinder [18] (JPF) is a software model checking tool for the Java language. Java Pathfinder can directly analyze the bytecode of Java multithreaded applications, checking the truth of assertions or LTL formulas. Java Pathfinder consists of a particular Java Virtual Machine (JVM) which executes the bytecode by exploring all possible execution paths (when nondeterministic choices are possible in the execution, each one of them is explored by backtracking execution).

JPF includes several optimizations that automatically reduce the number of states to be visited (avoiding those whose inspection is redundant) and thus the complexity of the analysis.

```
public class Client extends spiProcess {
 ...
 public void doRun(Channel cClientStart, Channel cClientServer1,
    SharedKey sk) throws SpiWrapperSimException{

 final Pair<Integer,Integer> pIdVal = cClientStart.receive(Pair.class);
 final Integer id = pIdVal.getLeft();
 final Integer val = pIdVal.getRight();

 event("setupRequest",val);

 final Nonce n = new Nonce();
 final Pair<Pair<Integer,Integer>,Nonce> pIdValN = new Pair<Pair<
    Integer,Integer>,Nonce>(pIdVal,n);

 final SharedKeyCiphered<Pair<Pair<Integer,Integer>,Nonce>> skc = new
    SharedKeyCiphered<Pair<Pair<Integer,Integer>,Nonce>>(pIdValN, sk);

 cClientServer1.send(skc);

 final Hashing h = new Hashing(pIdValN);
 final Hashing rH = cClientServer1.receive(Hashing.class);

 if(rH.equals(h)){
  event("finishRequest",val);
 ...

}
...
```

Figure 2.2. Excerpt of a JavaSPI model

Chapter 3

Spi2JavaGUI

3.1 Introduction

In Spi2JavaGUI, model driven development is leveraged to hide the complexity of a full implementation of a security protocol during the design phase, so that the developer only needs focus on a simplified abstract model. During this phase, formal verification is used in order to get assurance about logical correctness. In the implementation derivation phase, automatic generation of substantial parts of the implementation gives good assurance that the code adheres to the verified model. At the same time, the automatically generated code can be made immune from some kinds of low-level programming errors, such as buffer overflows, that could make the program vulnerable, but that are not represented in abstract models, and thus not captured by formal verification.

Spi2JavaGUI is implemented as an Eclipse plugin for the visual modeling of security protocols[1]. The framework offers an model driven development paradigm and integrates code generation and formal verification into one consistent and unified graphical interface. To make the approach affordable for non-experts in security, automated, non-interactive formal verification and code generation techniques are exploited.

The visual modeling approach of Spi2JavaGUI is the result of applying *data-driven* visualization inspired by well-known and successful modeling frameworks like Mathworks' Simulink to the security protocol domain. As a result, the user visually represents the flow of data between the principals of a protocol in a way similar to message sequence charts (MSC), which makes protocol modeling intuitive, and facilitates model understanding.

The proposed approach proved simple, yet expressive enough to model complex,

[1] Available as an Eclipse update site at `http://spi2java.polito.it/gui/updates`

real protocols. For example, a minimal but interoperable SSH implementation (included in the Spi2JavaGUI distribution) was developed with Spi2JavaGUI. However, for brevity, and to keep the exposition focused on the main fundamental aspects, only a smaller RPC example will be shown.

Spi2JavaGUI contributes on advancing the state of the art on visual modeling of security protocols, since none of the existing approaches integrates intuitive visual modeling with formal analysis and sound generation of interoperable code for the whole class of security protocols.

3.2 Spi2JavaGUI

The Spi2JavaGUI approach is an enhancement of the Spi2Java (see Section 2.2 approach that reuses the core of Spi2Java. As shown in Figure 3.1, both the spi calculus model and the refinement document are edited jointly in visual form, which represents one of the contributions of this work. Code generation, instead, is accomplished by re-using parts of the Spi2Java framework, which offers a sound code generation technique.

One of the main fundamental drawbacks of Spi2Java, which is solved in the work being presented here thanks to visual modeling, is that using Spi2Java requires excessive expertise because the user has to write models directly in the domain-specific spi calculus.

Furthermore, in Spi2Java the user is responsible for correctly following the implied workflow, which may lead to errors. For example, forgetting to run formal verification before code generation can lead to insecure code, while giving a false sense of security due to automatic code generation having taken place. This problem is solved in Spi2JavaGUI by having an integrated development environment (IDE) guiding the user through the different steps of model verification, refinement and implementation generation.

Finally, having separate documents for model and refinement information leads to model maintenance problems in Spi2Java, because even small changes in the spi calculus model may require the user to redo the full refinement step from scratch, due to the loose coupling between a model and its associated refinement document. This problem is solved as well in the proposed approach, because abstract protocol description and refinement information are strongly coupled in the visual model, without compromising its readability.

3.2.1 The model

A Spi2JavaGUI protocol model has the same structure and semantics as its corresponding spi calculus model, albeit with a different syntax and with some extensions.

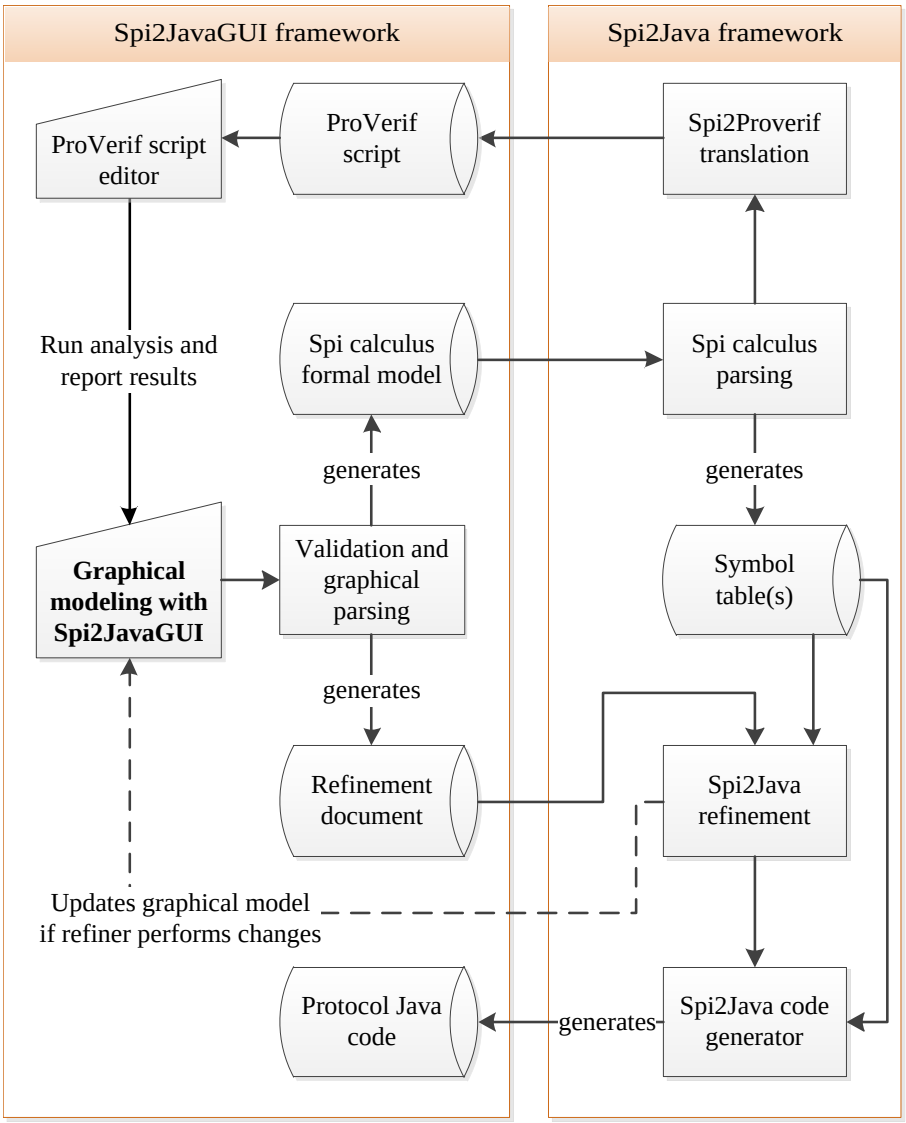

Figure 3.1. Workflow of Spi2JavaGUI and Spi2Java.

Some of the extensions provide the refinement information necessary for code generation, while other extensions provide visual information, necessary to visualize model elements. There is also the constraint that each process in the Spi2JavaGUI model must have a sequential behavior. This is not very restrictive because each process represents the behavior of one protocol role in a single protocol session, which is normally sequential.

Figure 3.2 shows a partial (and simplified) metamodel of Spi2JavaGUI. The classes named *Process*, *Term* and *Operation* (and all its specializations) are the same entities as found in spi calculus, while a *Diagram* represents a whole spi calculus

specification.

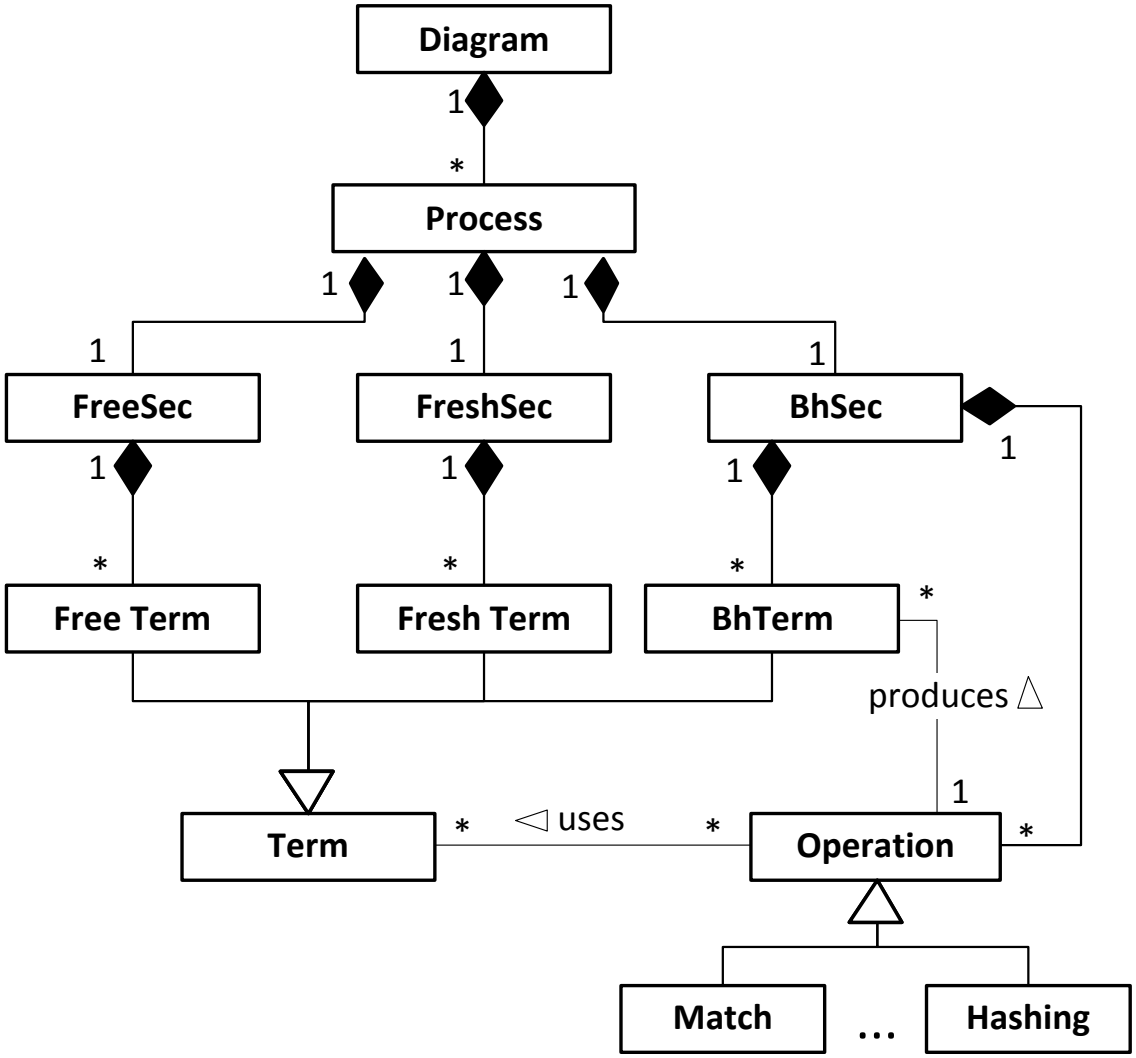

Figure 3.2. Simplified metamodel of Spi2JavaGUI.

Terms are divided into three classes, according to a classification that can be found in spi calculus as well. A *Free Term* is either a constant or a parameter used by the process, while a *Fresh Term* is a term that is created fresh during each execution of the protocol. A *Bh Term* is any other term that is produced as a result of an operation (e.g. the result $H(x)$ of hashing term x). Operations are ordered, according to their execution order in the sequential process they belong to. In order to keep the metamodel as simple as possible, and to reduce the efforts during the translation to spi calculus syntax, three *sections* (Figure 3.2 *FreeSec*, *FreshSec* and *BhSec*) have been introduced to separate different classes of terms and operations inside a process.

Some operations introduce choices, which leads to multiple possible execution flows. For example, a pair splitting operation is successful when the input term it uses is actually a pair, or fails when its input is not a pair, thus defining two possible execution flows. The behavior for each branch is specified inside two different *scopes* (not shown in the metamodel of Figure 3.2) that are associated with the operation.

Graphic and refinement information is coupled strongly with model elements (term, operation, etc.), by incorporating it inside each model element.

A Spi2JavaGUI model is scalable, because hierarchical models can be defined. This is achieved by introducing a special *container* operation, which has a custom number of inputs and outputs, and can implement arbitrarily complex behaviors, possibly using other nested container operations. In this way, the complexity of the internal behavior is hidden in a single operation block when observing the model at high abstraction levels, and users can design the model by first defining sorts of function interfaces and then writing their bodies later.

A number of consistency rules are defined on the model. For example, a process performing an input operation that uses a term, with no other process performing a corresponding output operation that produces the same term is considered a modeling error, which can be detected by a model validation procedure.

Given the strict correspondence between the Spi2JavaGUI metamodel and the spi calculus language, the formal specification in spi calculus deriving from the syntax translation of a Spi2JavaGUI model actually provides the formal semantics of the model.

3.2.2 Visual syntax

As observed by Selic [1], in order to best exploit the advantages of MDD, models should have some key features, among which *abstraction* (hiding detail for a given viewpoint, which lets one understand the essence easily), and *understandability* (using a notation that directly appeals to user intuition), In order to achieve abstraction, but at the same time understandability, the visual notation proposed here follows the approach of representing the different aspects of a security protocol in a *single* diagram.

The diagram is based on a *block-oriented* and *data-flow-oriented* view of the protocol, where principals and message flows are represented according to the common protocol intuition of message sequence charts. Blocks are used to represent operations, and the flow of message creation, transmission, and processing that takes place in each single protocol session is visualized as a chain of connected blocks, where each connection corresponds to a term (see Figure 3.4).

Moreover, to make this diagram immediate to read and easy to understand even with complex models, abstraction is used to hide non-relevant aspects, when one is focusing on a specific aspect of the protocol.

Specific aspects are identified and abstraction techniques developed, exploiting the specific security protocols domain. For instance, the message exchange scenario can be viewed, while hiding internal details of each principal. Conversely, drilling down on the same model, one can focus on the cryptographic algorithm to be used for a specific encryption operation in a principal, while hiding all other principals from the view.

The visual modeling and abstraction techniques are implemented in a prototype editor, which is part of the Spi2JavaGUI framework. As this editor is a prototype proof of concept, some aesthetic aspects have not been refined; nevertheless the editor is functional and implements all the abstraction techniques.

The attributes of a block can be set in a *property sheet*. Many of them are related to refinement information, only used at later stages, to obtain the final implementation. All attributes have default values, which lets the user simply neglect attributes that are only useful in later stages.

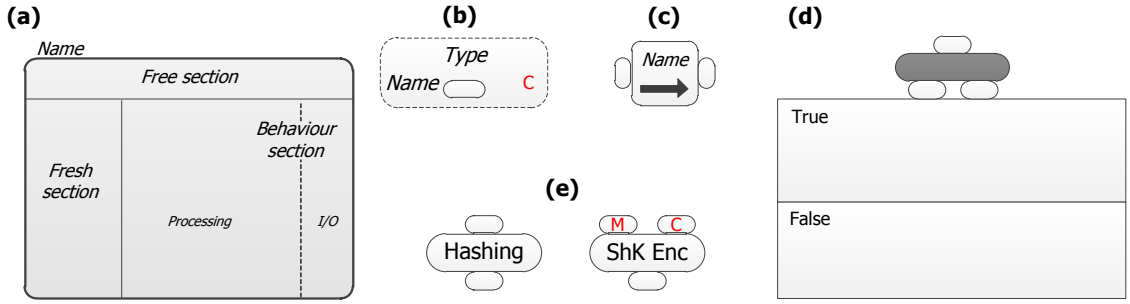

Figure 3.3. Some elements of the Spi2JavaGUI syntax: (a) process; (b) free term; (c) I/O block; (d) scopes; (e) computation blocks.

A process is visualized as a rounded corner box divided into sections (Figure 3.3(a)), according to the metamodel. The name of the process is on top of the box. Terms appearing in the free terms section are represented as in Figure 3.3(b) (where the red "C" indicates this is a constant term).

The *behavior* section is used to model the operations that follow one another in the process, and is itself divided into two subareas, not visibly marked but recognized as distinct for block placement. The main subarea contains all the blocks representing processing operations while the other subarea, which occupies the margin side non-adjacent to the fresh section, is named *channel section* and contains blocks representing input and output operations on channels.

The type of each block is represented by symbols and labels inside the block box (Figure 3.3(e) shows a hashing and a shared-key encryption). Each block has input and/or output ports. Input ports are positioned on the upper side while output ports are positioned on the lower side, with the exception of input/output blocks, which have ports on their left and right sides (Figure 3.3(c)).

When an operation block has two or more input ports that are not interchangeable (e.g. in encryption blocks, where plain text and key must be connected to the right ports), numbers or letters are placed near each port to identify the meaning of each of them. The same is done for output ports.

Some operation blocks have a variable number of input and/or output ports, selectable by the user according to necessity (e.g. *pair composition* and *container* blocks).

When a new input/output block is added, the user has to set the type (input or output) and the term that represents the channel. The direction of communication is represented by the arrow just below the channel name, which points outwards for output and inwards for input.

Scopes are represented as shown in Figure 3.3(d). They are created automatically just below each operation block that implies a choice. Any scope can be collapsed or expanded according to the details the user is interested in. For example, error handling in a protocol can be easily abstracted by collapsing else branches of failed matches. When a scope does not contain any operation, this simply means that the execution of the process stops in that case (in the generated Java program, an exception will be thrown).

In order to correctly map the diagram onto the model, some rules are necessary. For example, connections can be placed only from an output port to an input port. An output port can be the "source" of many connections, but an input port can accept only a single connection. The only blocks that can accept or originate a connection from/to outside the process are the input/output blocks.

On each connection a label shows textually the symbolic value of the term that is conveyed by the connection.

Connections establish dependency relations between blocks of the model: the output value(s) of an operation depend on the block(s) that are connected to its input port(s). These relations imply ordering constraints for operations: if operation A produces a term used by operation B, then the execution of A must precede the execution of B. Then, for processing operations, ordering is determined following connections backwards, while the position of blocks is irrelevant. This design decision allows the user to freely organize operation blocks to make the diagram readable, while their ordering will be automatically computed.

In contrast, time flows from top to bottom in the area that contains the input/output blocks, so that the position of each input/output block determines the order of message exchanges. In this way the user gets a linear representation of the protocol message exchange, much like in widely used message sequence charts.

An algorithm is run to automatically compute the order of operations implied by these rules whenever some change occurs. This amounts to convert the data-flow representation that is displayed, into the control-flow one of the underlying model. If more than one ordering of operations is compatible with the constraints deriving

from dependencies and positioning, the final ordering is determined deterministically by applying default ordering choices. In this way, the model always includes, for each process, a fully ordered set of operations, which can be directly translated into spi calculus syntax.

3.2.3 The RPC example

A simple authenticated Remote Procedure Call (RPC) protocol will be used as a running example to show different features of the Spi2JavaGUI framework. The essence of this protocol will be first introduced informally, by using a simple notation that is usually adopted in the security protocol community to represent the main sequence of message exchanges. Then, it will be shown how the full protocol can be formally modeled and verified in the Spi2JavaGUI visual framework, and how an implementation can be derived.

Note that the informal description that will be given shortly to introduce the protocol does not describe several aspects of the protocol. For example, the handling of error situations (e.g. when a message of the wrong type is received) and the operations performed on received messages are not represented.

The protocol involves two principals, named Alice and Bob (A and B in the abbreviated notation). The security goals of the authenticated version of the RPC protocol are that whenever a principal B accepts a request message from A, principal A has really sent the message to B and, conversely, whenever A receives a response message from B, principal B has really sent the message in response to the matching request from A.

The protocol is request-response with the two main messages described in the abbreviated notation as follows.

1. $A \rightarrow B : S, Na, H(K_{AB}, REQ, Na, S)$
2. $B \rightarrow A : f(S), H(K_{AB}, RES, Na, S, f(S))$

The request message goes from Alice to Bob and contains a string S, which identifies the remote procedure to call and its parameters, a cryptographic nonce (i.e. a fresh random number) N_a, and a keyed hash (or HMAC), calculated as the cryptographic hash of the concatenation of a shared key K_{AB}, a tag (constant string) REQ, the nonce N_a and the string S.

The response message, which flows from Bob to Alice, is composed of $f(S)$, which represents the output of the remote procedure call, and an HMAC calculated with the shared key K_{AB}, the RES tag, the nonce N_a, and the request and response strings. For simplicity, in this example $f(S)$ will be defined as the cryptographic hash of S, but in general this can be an arbitrary function.

Please note that this protocol enjoys only some of the desired security properties, which will be discussed later on, in the part about formal analysis (see Section 3.2.5).

Figure 3.4 shows the complete visual model of both principals in full detail, including handling of error conditions.

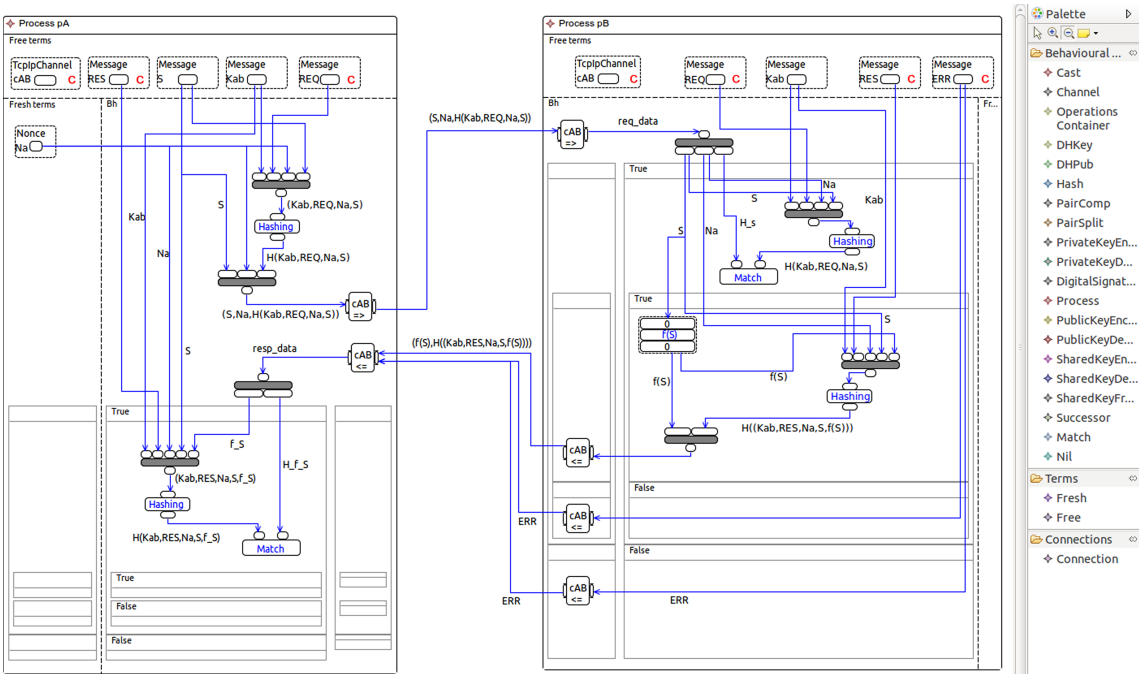

Figure 3.4. Complete model of the RPC protocol.

3.2.4 Model validation

In addition to making modeling simple and intuitive, the visual interface also enforces the internal consistency of the model during its development. Some possible mistakes are directly avoided during editing, because the editor does not allow inserting invalid objects, values or connections (e.g. it is not possible to create loops of connections, or to place an operation block inside the sections reserved to terms).

Validation of the modeled protocol is automatically performed each time the user saves a model, and before running ProVerif analysis or code generation.

The validation process is based on OCL-like rules and looks for all those problems that cannot be avoided with the in-editing checks. There are two types of such problems that are reported: *warnings* (denoted by yellow triangles with exclamation marks) and *errors* (denoted by red circles with white cross), as depicted in Figure 3.5. A warning indicates a problem that does not prevent formal analysis or code generation, for example an unused output of a block. An error indicates a problem that must be resolved before starting formal analysis or code generation (e.g. a missing input connection in a port of an object). Warnings and errors are also logged in the Eclipse logger, and each of them contains a message with details about the problem.

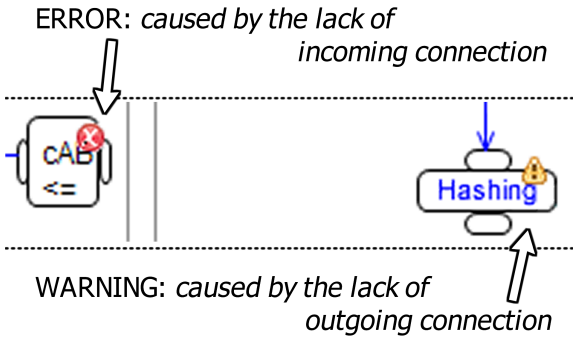

Figure 3.5. Spi2JavaGUI: example of error and warning

3.2.5 Formal analysis

Once the protocol is modeled, and the validation phase passed, the user can run the ProVerif (see Section 2.1) verifier to perform formal analysis. Indeed, formal verification requires that the security properties of the protocol are formally specified. This specification step can be integrated in the protocol modeling visual interface. For example, secrecy of a free term can simply be an attribute of the term block, while authentication properties, that are commonly expressed as agreement or correspondence relationships, can be represented by special blocks connecting the terms upon which agreement is required and the operations upon which correspondence is required[2].

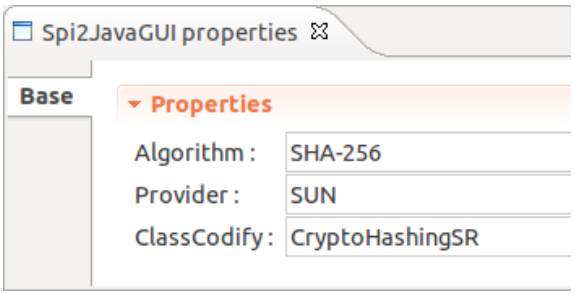

Figure 3.6. Spi2JavaGUI: property sheet

Note that formal verification does not require that implementation details have been specified with their final values, because formal verification just uses the abstract model and disregards implementation details. Therefore, abstractly specifying and formally verifying a protocol model is as simple as placing and connecting blocks

[2]In the current implementation of the prototype tool, this part has not yet been implemented, so that properties have to be written in ProVerif syntax.

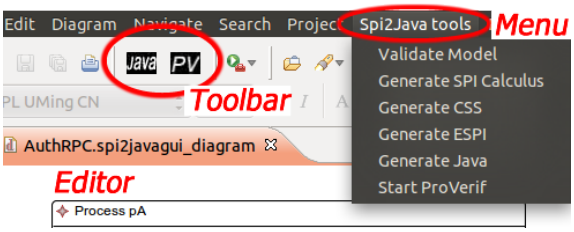

Figure 3.7. Spi2JavaGUI: toolbar and menu

in the editor, saving the model and running the formal verification tool. Property sheets (Figure 3.6) can be neglected in this phase, which confirms the ability of Spi2JavaGUI to work at different abstraction levels.

The formal analysis process, which can be started simply by a button click, (Figure 3.7), automatically translates the model into a formal specification, expressed in spi calculus using the syntax accepted by the ProVerif verification tool, and runs the tool, after having included the specification of the properties to be verified (which at present is user-provided in text form).

Moreover, in order to analyze the behavior of the protocol with any number of concurrent sessions, an additional process that instantiates a possibly unbounded number of concurrent copies of the various protocol processes is also generated.

In the RPC example the security properties that will be (tentatively) verified are the secrecy of terms S and Kab (meaning that a Dolev-Yao attacker is unable to know their values), and the agreement [24] upon the request message (the meaning of agreement is explained below in the results). Figure 3.8 shows the result of the ProVerif analysis in textual form (as shown in the current prototype).

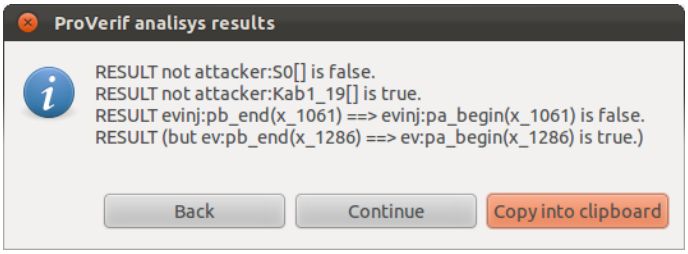

Figure 3.8. Result of ProVerif analysis performed on the example protocol.

From the results given by ProVerif it is possible to claim that:

- term S can be known by an attacker;
- term Kab remains secret;
- the protocol does not fulfill the injective agreement property upon the request

message (this is because an attacker can replay the request message to process pB, which accepts the message as if it had been truly sent by pA);

- the protocol fulfills the non-injective agreement (weaker than injective agreement) property upon the request message (this means that if pB believes it has received a request message from pA once or more, then pA previously really sent that message at least once). In other words, an attacker cannot forge new request messages from scratch and get them accepted as authentic.

Beware that this is only a simple formal analysis presented as an example. In real scenarios more queries and events may have to be defined to verify all the security properties of a protocol.

3.2.6 Code generation

The last development phase is the generation of concrete implementation code in the Java language. Before starting this phase, the protocol must have been fully modeled and refined, including implementation details which must have been specified to their final values. Moreover, the validation process must have been executed without errors.

In Spi2JavaGUI, code generation is performed by the user via a single push-button operation (see Figure 3.7). Internally, some Spi2Java libraries are re-used, to achieve a four-steps code generation process:

1. the generation of spi calculus from the visual model;
2. the generation of a symbol table for each process;
3. the generation and checking of a refinement document for each process;
4. the generation of the final Java code that implements the protocol.

These complex interactions with the Spi2Java libraries are fully abstracted by the workflow of Spi2JavaGUI. Especially, refinement information is now embedded in the graphical model (via the property sheets), and so implementation generation can happen in a single, automatic step.

Using the Spi2Java code generation engine to obtain the final code brings an added value, i.e. the formally proved soundness of the code generation process. More precisely, as shown in [23], the generated code is guaranteed to fulfill security properties such as secrecy and authentication under a Dolev-Yao attacker, provided that the same properties hold on the abstract formal model from which the code has been generated. Since security properties are formally verified on the abstract model before generating code, high confidence on the generated code is finally obtained.

3.3 Related work

Some approaches have already been developed to address the modeling, formal verification and implementation of security protocols. In this section the focus is mainly on existing visual modeling formalisms that combine model-driven development and formal verification.

3.3.1 Custom Visual Formalisms

McDermott [25] presents GSPML, a visual modeling formalism intended to be a rigorous base for formal verification and to address at the same time some deficiencies perceived to exist within other visual modeling languages.

A model that uses the GSPML formalism defines implicitly all the possible traces of an instance of the protocol, which includes not only the expected normal behavior, but also the behavior when under the attack of a Dolev-Yao intruder. The GSPML approach specifies only a graphical formalism, not a corresponding textual process algebra. However, McDermott reports that GSPML can be used as a front-end to visually represent security protocol models of both the classical CSP [26] and the PEPA [27] process algebras, which enables formal verification.

Hence, the strength of GSPML is its use of visual models to formally represent high-level protocol specifications in a concise manner, aiding in both understanding and verification. The main deficiency of GSPML, in view of model-driven development, is that protocol models are event-based and trace-oriented, thus omitting internal computation aspects, which are essential for protocol implementation. As a consequence, there is no provision for automated and rigorous transition from a GSPML model to code, which makes GSPML more oriented to protocol analysis by security experts than to protocol implementation by software engineers.

From the point of view of visual modeling, GSPML makes extensive use of textual notation to describe the structure of events and messages. For this reason, GSPML exploits visual modeling to a lesser extent than the approach presented in this thesis.

Another approach, named SPEAR II is described by Saul and Hutchison [28]. They propose visual modeling of security protocols integrated with the GNY formal analysis tool, which is based on a variant of the BAN modal logic [29]. Visual modeling is based on MSC-like diagrams where only the main sequence of messages exchanged in a normal run of the protocol is represented. While this representation is enough for a BAN-logic analysis, the gap between it and a protocol implementation is too large, thus making this kind of model not suitable for a rigorous transition from model to protocol implementation.

3.3.2 UML-Based Security Modeling

UML is widely used in software design, as a standard for object-oriented modeling, however it lacks formal semantics and it does not provide specific features to treat security aspects. Some work has been done, trying to extend UML in order to make it possible to model security aspects, and providing UML subsets with formal semantics.

Some initiatives to use UML for security aspects came from Epstein and Sandhu [30] and Basin et al. [31]. However, these UML extensions are for specifying role-based access control policies rather than security protocols. Similarly, Bushager and Zwolinski [32] exploit SystemC with the Transaction Level Modelling extensions [33] to create executable models starting from UML models. The executable models give the opportunity to see the animations of transaction flows defined in the initial UML specifications. Attacks in different parts of the system can be simulated within the executable models. Nevertheless, this solution can be applied to all UML models and does not provide security-specific features, i.e. the user has to define attacker capabilities and security properties to be verified during simulation.

Jürjens [34] proposed UMLsec, which allows annotation of UML diagrams (through stereotypes, tagged values and constraints) to express security requirements (e.g. confidentiality and integrity of messages). These can be then validated using formal verification techniques. UMLsec has also been proposed as a means for visual modeling and formal verification of security protocols [35]. Jürjens also showed a way to systematically generate test sequences for security properties specified in UMLsec models, so these sequences can be used to test implementations for vulnerabilities [36]. The UMLsec-based visual modeling of a security protocol is based on annotated sequence charts, which has similarities with the approach proposed here. In more detail, the protocol message sequences are described through sequence diagrams, while class diagrams describe the state of the agents and contain annotations that specify security relevant information.

The main differences with respect to the approach proposed here is that messages are represented only textually, and implementation details cannot be added in the visual model. As a consequence, the UMLsec approach is good for protocol formal analysis but it does not support rigorous, traceable and automatic transition from design to concrete interoperable code. Extending UMLsec to support automatic code generation would technically be possible. However, this would require the addition of new views, with the inherent problem of keeping them synchronized. Moreover, the addition of such views to the several ones already existing would make the protocol representation too fragmented, and overall difficult to understand. Hence, the visual modeling being presented in this work is designed to offer a single unified view, where view-collapsing is used to hide non-relevant details.

SecureMDD [37, 38, 39], proposed by Moebius et al., is another approach based on UML for the representation of security protocols. This approach is closely related

to the one proposed in this chapter because it uses the concept of model-driven development and enables formal verification.

Unlike Spi2JavaGUI which is targeted to any security protocol, SecureMDD is an application-specific approach for introducing security aspects into smart card systems: design models can be translated only to Java Card code and only smart card domain specific properties can be verified.

From the UML model both a formal model based on state machines and an interoperable Java Card implementation can be obtained. However, differently from Spi2JavaGUI, no formal soundness proof relating the formal model and the generated code is available. Another difference is that SecureMDD requires the definition of several different models (containing platform-independent and platform-specific information respectively) to obtain the implementation and the formal model while Spi2JavaGUI unifies the abstract view and the implementation details into a single consistent graphical view and model.

Smith et al. [40] developed a modeling technique using UML 2.0, without extensions, that exploits the *ports* and *protocols* features to define the communication between participants. The UML 2.0 infrastructure also provides the ability to create an executable model and to generate executable code. However, no formal semantics is provided for the visual model, so formal verification cannot be supported. Indeed, the user is responsible for modeling and executing possible threat scenarios against the protocol. This can be used to ensure the designed protocol is resilient to known flaws, but cannot be used to identify new ones. Lack of formal semantics also implies that no soundness guarantees can be obtained for the generated code.

Chapter 4

Automated formal verification of application-specific security properties

4.1 Introduction

The issue of formal verification of distributed applications has attracted the interest of a considerable number of researchers in recent years. In fact, over time, applications that communicate via networks have become more complex, and at the same time also the security requirements of those applications have evolved and become more complex.

Distributed applications use cryptographic protocols to communicate securely over insecure channels. Generally, an insecure channel is a communication channel that does not guarantee, by itself, the security (e.g. privacy, integrity, and authenticity) of the data that are exchanged among the processes of the application that uses the channel. Examples of insecure channels include Internet, radio channels (e.g. Wi-Fi) and local networks, if connections can be tampered with by malicious attackers. Security protocols exploit cryptographic primitives to ensure the desired security properties when communicating over insecure channels.

Nowadays, solutions like the Transport Layer Security (TLS) protocol can be used to deploy security in distributed applications, in a stable and reliable way. With this approach, the security protocol and the application logic layers are totally independent: the protocol guarantees standard security properties (mutual authentication, confidentiality, data integrity), by introducing a new software layer under the application, and the application is developed in a nearly security-unaware way, security being provided just by application insulation, which is guaranteed by the fact that the application communicates only using the functionalities provided by the security protocol layer. In other cases, however, protocol and application

logic are less independent. For example, if the application logic needs to interact more strictly with the protocol, in order to guarantee application-specific properties, custom protocols may be used. Of course, using an independent security layer implemented by standard protocols (for instance TLS) is preferred when possible, because of its simplicity, interoperability and reliability. However, this is not always possible or convenient, for example because the devices involved do not have enough hardware resources or do not have standard connectivity to the Internet, but only limited ad-hoc connectivity.

In the past, the techniques and tools for automated formal verification mostly targeted to either the analysis of security communication protocols or the analysis of application code. For example, the tool proposed by [41] can formally verify standard security properties of cryptographic protocols under the presence of active attackers. However, such tools can analyze only the bare protocol (message exchanges and related checks) while they are not adequate to also model and analyze the application logic that interacts with the protocol, which can be made of complex programs, developed without particular constraints. Generally, the tools that belong to this family cannot deal with application-specific security properties. Instead, tools for the automated formal verification of arbitrary application source code(e.g. software model checkers [18]), can check even complex requirements, directly on the implementation of the application. In theory, these tools even allow considering active attackers in the system, but a model of those attackers must be supplied, and the inclusion of active attackers increases verification complexity considerably.

However, scalability of these approaches is the main obstacle to extending existing verification techniques to analyze both protocol and application logic together in the face of active attackers. In fact, the problem of cryptographic protocol verification is itself challenging despite the simplicity of such protocols, and application code can be very large and complex.

The idea that is developed in this chapter of the thesis is that, instead of extending either of the currently available verification techniques, a convenient way for enabling the verification of whole distributed applications made of security protocols and application logic is to exploit already existing verification techniques jointly, by performing assume-guarantee compositional verification. An innovative methodology was presented in [5]. It combines two already existing and well-known automated formal verification techniques, *theorem proving* for cryptographic protocol verification and *model checking* for source code verification, according to the principles of assume-guarantee compositional verification. This approach brings better scalability, due to the splitting of the verification problem into simpler sub-problems. Moreover, the approach described here approach lowers the barriers for non-security-expert developers, because is based on verification techniques that are automated, simpler to use, and that can also provide counter examples when the properties to be verified do not hold. Finally, this solution also aims at automating the entire process of

implementing and verifying distributed applications.

More specifically, the proposed development adheres to the principles of model driven design: the first step consists of the definition of a high-level formal model of the communication protocol, which also contains the declaration of the expected security properties of the protocol. The automated formal verification of those properties is performed by the protocol verifier ProVerif (see Section 2.1), and the Java implementation of the protocol is automatically generated by the model driven development framework JavaSPI [3] (see Section 2.3), which guarantees the preservation of the intended security properties. Then, the generated protocol implementation has to be integrated within the application logic (both client(s) and server(s)), which can be developed in any way (hand written or developed using other code generation techniques). The application-specific properties are formulated and verified on the final implementation using a Java source code verifier, such as Java Pathfinder (JPF)[18], but taking the results of the protocol formal verification into account. This is achieved by replacing the code that implements the protocol (generated with JavaSPI), with a stub that enforces the properties already verified on the protocol model. The main advantage of the methodology proposed here is the reduction of verification complexity, if compared to a separate and independent use of the theorem prover and the model checker, achieved by leveraging compositional verification in the assume-guarantee reasoning style.

The development workflow is largely automated: this helps to reduce the probability of introducing errors, and enables quick error diagnosis (both the tools used for formal verification phases provide counter examples, i.e. the execution traces that violate the requested security properties).

4.2 Related work

In the last decades many automated techniques have been developed for the formal analysis of security protocols, as surveyed in Patel et al. [41].

These techniques analyze high-level abstract models, in order to prove the correctness of the protocol logic. More recently, some researchers have started working on techniques that bring automated formal proofs closer to real implementations of security protocols [14], [42]. Among these are the model-driven development approaches, like the one exploited in [3].

All the above mentioned techniques are focused on security protocols rather than on whole applications, and address the generic security properties enforced by such protocols (e.g. authentication, secrecy and integrity), rather than the application-specific security properties.

Some papers have addressed the formal verification of security protocols for specific applications, such as for example electronic commerce, with their related application-specific properties. Surveys about techniques in this field are given by

Ouchani and Debbabi [43], and Nguyen et al. [44].

Bella et al. [45] presented the formal verification of some application-specific properties of the suite of protocols "Electronic Secure Transaction", used for e-commerce. However, this work is substantially different from the one presented here because verification is not automatic (being based on the interactive theorem prover Isabelle [46] which requires human assistance), and what is formally verified is only an abstract model of the application rather than its final implementation.

Moebius et al. [38] and Borek et al. [39] presented two case studies of formal verification of application-specific security properties (i.e. the truth of a predicate involving some variables of the application), taking into account both the protocol and the application logic together. In these case studies the application is developed with a model-driven approach and the model is used to generate a formal specification, which afterwards can be verified by an interactive theorem prover. An important limitation of this approach is that it is based on interactive theorem proving, which is not automatic, is very time consuming, and requires a lot of expertise. Moreover, if the application is flawed, interactive theorem proving does not provide counter examples, which can make error diagnosis and correction very difficult. A related publication [47] presents exactly the same methodology but applied to a service-oriented application. In addition, some other papers have addressed the problem of developing distributed applications with formally verified security properties. A recent paper [48] extends the previous approach by integrating the AVANTSSAR [49] model checker into SecureMDD. As a result, it is possible to automatically generate a formal specification for the model checker from a UML model. However, only some application-specific properties can be verified using AVANTSSAR. For example, differently from the work presented here, which enables the verification of arbitrary properties, it is not possible to compare numeric values inside the model checker.

Jürjens [35] proposed a UML-based technique for the specification of distributed applications and automated formal verification of application-specific security properties. The technique was applied to the Common Electronic Purse Specifications regarding payment via smart-card. One of the properties that were verified is, for example, that the amount of money in the system is every time the same, that is the total sum of budgets of smart-card holders is always equal to the sum of the earnings of all merchants. However, this technique provides formal verification of UML models only, whereas a formal link with the application implementation is missing. Moreover, differently from the approach described here, verification is performed in a single step on the whole model, without using compositional verification.

Gunawan et al. [50] proposed a method to integrate some standard security mechanisms (for protecting information transfer) into distributed applications automatically. The paper includes a proof that the security mechanisms are integrated into the application so as to fulfill some generic properties. However, this approach does not target the verification of application-specific properties.

The idea of using compositional verification to formally verify application-specific security properties of distributed applications already appeared in Gunawan and Herrmann [51]. In that work, however, formal verification is done by a general-purpose model checker, without considering active network attackers and the properties of cryptographic operations.

Finally, Vasilevskaya et al. [52] proposed a formal domain-specific language approach for the development of security-enhanced embedded networked applications. The domain specific language, which defines applications as compositions of building blocks, provides a bridge between security domain experts and embedded domain experts. However, this methodology is limited to embedded applications built with specific blocks, and does not enable the verification of application-specific security properties.

4.3 The extended JavaSPI

To achieve the final goal of this work the JavaSPI framework has been extended in order to enable increased interaction between the generated protocol code and the application that uses the protocol. With the original JavaSPI, only a simple interaction mechanism was possible, where the application starts a protocol session, passing input arguments, and, upon termination of the protocol session, the application gets the outputs. With the extended JavaSPI version, the application can be called back by the protocol code when some events defined in the model occur. In this way, the application can receive outputs from the protocol at intermediate stages of a protocol session. The `@EventsInterface` annotation enables this new mechanism. When the annotation is present, the code generator generates a Java interface that contains the methods associated with the events generated by the process and has the name specified in the annotation. Events are defined inside the JavaSPI model by means of the instruction `event` (see details in Section 4.3.1). When a session of the protocol is started by the application, a callback object that implements the generated interface must be passed as argument. This extension does not affect the validity of the ProVerif model that is generated by translating the JavaSPI code, because the methods called on event occurrence are one way notifications that cannot alter the protocol behavior (modeled and verified by ProVerif). As detailed in Section 4.6.6, when performing the verification of the application code, the protocol code is substituted by stubs that enforce exactly the same event orderings that are allowed by the protocol.

4.3.1 Matching events in the model and methods in the code

The `event` instruction of JavaSPI models has a variable number of parameters (at least one). The first parameter is a string, and represents the name that JavaSPI

associates to the event instruction. The following parameters, if present, must be objects, previously created in the protocol, that extend the `Message` class. The `event` instruction is translated in the concrete Java implementation of the protocol to a call of a method, that has the same name of the event, and the corresponding concrete objects are passed as arguments. The method belongs to a class that implements the interface defined in the `@EventsInterface` annotation of the JavaSPI model.

For example, if the JavaSPI model contains the following code:

```
@EventsInterface("InterfName")
public void doRun(Channel ch, ...){
  ...
  final Integer iVal = ch.receive(Integer.class);
  ...
  event("evtOne",iVal);
  ...
}
```

The resulting concrete protocol Java code is:

```
public Map<String,Message> performHandshake(Channel ch, ..., InterfName
    InterfName_impl){
  ...
  final Integer iVal = ch.receive(Integer.class);
  ...
  InterfName_impl.evtOne(iVal);
  ...
}
```

In this way, every `event` instruction in the JavaSPI model has a corresponding invocation of a specific method in the concrete code, along with the corresponding arguments.

4.4 Translation rules

This section specifies how, starting from a JavaSPI model, it is possible to automatically generate the stubs that replace the protocol behavior during the formal verification of the Java application code.

The stubs are built in such a way that they can generate the same sequences of events that may be generated by a real execution of the protocol. As these sequences are constrained by the correspondence properties that have been formally verified on the security protocol, the stubs are built out of such properties.

Correspondence properties (also called queries) are located inside a `@PStubQueries` annotation, which is applied to the master class of a JavaSPI model (the master class is a special actor class that instantiates the roles of the protocol with the right arguments). Figure 4.1 defines the grammar for correspondence properties accepted by JavaSPI. In Figure 4.1, non terminals are written in italics and terminals in

normal font. The root element is *queries*. Non terminal *event-name* corresponds to the name of an event defined as first argument in one of the JavaSPI `event` instructions contained in the model. The conjunction (`&`) has higher priority than the disjunction (`|`), but parentheses can be used to disambiguate the expressions. The `inj:` prefix is used to specify that the correspondence is injective, i.e. the event after the `==>` symbol must have a 1:1 correspondence with the event defined before the `==>` symbol. The prefix `inj:` is automatically added to the event before the `==>` symbol, if the event after the `==>` symbol has the `inj:` prefix.

For simplicity of implementation, some restrictions regarding event declarations have been defined: (i) it is not possible to use the same event name with a different number or type of parameters in the JavaSPI model (the JavaSPI model parser does not allow "overloading" of events); (ii) the same event can be used in different queries, but it can be used before the `==>` in only one query; (iii) the same event name cannot be used in the `@PStubQueries` annotation both with and without the `inj:` prefix (in order to reduce the complexity of the stub generation process); (iv) the events contained in the `@PStubQueries` annotation must have as arguments, in the JavaSPI model, only values received from the application that calls the protocol through channels (e.g. it is not possible to use protocol nonces or keys as arguments, because these values cannot be abstracted from the model). However, these restrictions do not limit the flexibility of the approach and the possible queries that can be specified, but may require the definition of multiple events (with different names and/or different arguments) in the same point of the protocol (in the case of limitations (i) and (iii)), in order to verify all the properties. In some cases the limitation (ii) can be overcome by rewriting the queries in the following way: from the two queries `evt1 ==> evt2` and `evt1 ==> evt3`, to the query `evt1 ==> evt2 & evt3`. Finally, regarding limitation (iv), an application that uses a protocol generated by JavaSPI should not care about internal objects in the protocol. For example, if the protocol uses shared secret keys to protect messages, the application that uses the protocol must not access to that objects, because the protocol objects are not relevant for the logic of the application. Thus, the entire software (application and protocol) must be designed in order to respect this constraint. In any case, this does not limit the number and types of object that an application can use as arguments of events (except internal objects in the protocol), but prevents the application from accessing to internal protocol objects.

The first generated elements are the interfaces which contain the definitions of the events. Each actor (excluding the "master") of the JavaSPI model originates a different Java interface. The name and the package of the interface may be customized by using the `@EventsInterface` annotation. Eventually, it is possible to select a subset of all the events defined in the actor that will be considered in the interface generation (by default, all the events are considered). For example, the instruction `event("clientOperation",xValue);`, where `xValue` is an instance

```
queries          ::=   { non-empty-q-list }
non-empty-q-list ::=   query , non-empty-q-list
                     | query
query            ::=   fact ==> hypothesis
fact             ::=   event-name
                     | inj:event-name
hypothesis       ::=   fact
                     | hypothesis & hypothesis
                     | hypothesis | hypothesis
                     | ( hypothesis & hypothesis )
                     | ( hypothesis | hypothesis )
                     | ( fact ==> hypothesis )
```

Figure 4.1. Grammar for correspondence queries.

of the class `it.polito.javaSPI.spiWrapperSim.names.Integer`, originates the following method definition in the interface:

```
public void clientOperation(
  final it.polito.javaSPI.spiWrapperSim.names.Integer xValue
);
```

All the generated methods in the interfaces have no return value (i.e. `void` return type) because the application can only receive objects from the protocol, but not pass objects to the protocol code as return value of methods associated with the events. This characteristic ensures that the application does not alter the flow of the communication protocol by passing objects to the protocol. The application that uses the JavaSPI generated code must provide classes that implement each interface, before starting the Java Pathfinder analysis.

The second part of the generation process produces the `EventsManager` class (the destination package can be customized by using the `@PStubPackage` annotation in the "master" actor). The methods contained in this class enforces the constraints and precedences defined by the correspondence properties. In particular, the methods which name ends with `_notify` track the occurrences of the events with the corresponding object passed as arguments (for the injective events, it is necessary to count the occurrences for each different set of arguments). These methods use internal hash maps to keep track of the event occurrences. Similarly, the methods which name ends with `_isEnabled` are used to check if events had occurred previously (i.e. the `_notify` was invoked with the same arguments). A detailed description of the methods generated inside the `EventsManager` is given in the continuation of the section.

The queries contained in the `@PStubQueries` annotation are parsed in order to create four different sets, that contain the names of the events previously declared in the interfaces. The $\mathcal{R}_{NI}$ set contains the names of all the events defined in

correspondence properties queries after the `==>` symbol. Similarly, the $\mathcal{L}_{NI}$ set contains the names of all the events defined before the `==>` symbol. The same event may belong to both $\mathcal{R}_{NI}$ and $\mathcal{L}_{NI}$ sets. For example, the query `evt1 ==> evt2 ==> evt3` originates the following sets: $\mathcal{R}_{NI} = \{evt2, evt3\}$ and $\mathcal{L}_{NI} = \{evt1, evt2\}$. If the an event specified in a query has the `inj:` prefix, the event will be included into the $\mathcal{R}_I$ and/or $\mathcal{L}_I$ sets (instead of $\mathcal{R}_{NI}$ and $\mathcal{L}_{NI}$), following the same rule described above. However, it is important to remember that the prefix `inj:` is automatically added to event before the `==>` symbol, if the event after the `==>` symbol has the `inj:` prefix. For example, the query `evt4 ==> inj:evt5 ==> evt6` produces the following sets: $\mathcal{R}_I = \{evt5\}$, $\mathcal{L}_I = \{evt4, evt5\}$ and $\mathcal{R}_{NI} = \{evt6\}$.

The methods of the `EventsManager` class are automatically generated by using different templates for each defined set of *events*, presented here, where *EVT-NAME* is the name of the event, *TYPE-PAR-LIST* represents the parameters types and names list in the method declaration, and *PAR-LIST* is the list the name of the parameters in the method declaration (i.e. corresponds to *TYPE-PAR-LIST* without types). The type `Message` refers to the `it.polito.spi2java.spiWrapper.Message` class.

Each event contained in the $\mathcal{R}_{NI}$ is translated using the following template:

```
private static final HashSet<List<Message>> EVT-NAME = new HashSet<
    List<Message>>();

public synchronized void EVT-NAME_notify(TYPE-PAR-LIST){
  EVT-NAME.add(Arrays.asList(PAR-LIST));
}
```

Each event contained in the $\mathcal{R}_I$ is translated using the following template:

```
private static final HashMap<List<Message>, java.lang.Integer> EVT-
    NAME = new HashMap<List<Message>, java.lang.Integer>();

public synchronized void EVT-NAME_notify(TYPE-PAR-LIST) {
  final List<Message> list= Arrays.asList(PAR-LIST);
  final java.lang.Integer v = EVT-NAME.get(list);
  if (v != null) {
    EVT-NAME.put(list, v + 1);
  } else {
    EVT-NAME.put(list, 1);
  }
}

private boolean EVT-NAME_injDecrease(TYPE-PAR-LIST) {
  final List<Message> list= Arrays.asList(PAR-LIST);
  final java.lang.Integer v = EVT-NAME.get(list);
  if (v != null && v > 0) {
    EVT-NAME.put(list, v - 1);
    return true;
  } else {
```

```
    return false;
  }
}
```

The events contained in $\mathcal{L}_{NI}$ and $\mathcal{L}_{I}$ are translated using the following template:

```
public synchronized boolean EVT-NAME_isEnabled(TYPE-PAR-LIST) {
  return BOOL-EXP;
}
```

The *BOOL-EXP* value is replaced with a boolean expression which depends on how the *EVT-NAME* event precedences are defined in the `@PStubQueries` annotation. More precisely, the boolean expression considers the hypothesis defined after the `==>` following the *EVT-NAME* event. For example, if *EVT-NAME* is `evt8` and the query containing the event is `evt7 ==> inj:evt8 ==> inj:evt9 & evt10`, the hypothesis that will be considered is `inj:evt9 & evt10`. Similarly, if *EVT-NAME* is `evt7` the hypothesis is `inj:evt8`. The string *BOOL-EXP* (initialized to an empty string) is built using the following algorithm:

i the hypothesis is split into tokens (delimiters are: `(`, `)`, `&`, `|`), and the resulting array `T` contains both delimiters and tokens, in the same order as they appear in the hypothesis

ii the variable `i` is set to 0

iii the array element `T[i]` is parsed: if it is a delimiter, then it is copied as is to the *BOOL-EXP* string. Otherwise, if `T[i]` is the name of an event (e.g. named *evt_x*) without the `inj:` prefix, the string *evt_x*`.contains(`*PAR-LIST*`)` is appended to the *BOOL-EXP* string. Finally, if `T[i]` is the name of an event with the `inj:` prefix, the string *evt_x*`_injDecrease(`*PAR-LIST*`)` is appended to the *BOOL-EXP* string

iv the variable `i` is incremented by 1 and, if `i` is less that the size of the array `T`, the process continues at point (iii). Otherwise, the process is completed.

For example, if the hypothesis is `inj:evt9 & evt10`, the resulting *BOOL-EXP* string is `evt9_injDecrease(`*PAR-LIST*`) & evt10.contains(`*PAR-LIST*`)`, where *PAR-LIST* is replaced with the correct arguments.

The third and last part of the generation process consist of the translation from the JavaSPI abstract actor model to the corresponding stub. For each actor, a different Java class is generated. The generated class has a name that corresponds to the name of the actor in the JavaSPI model plus the `_EventsStub` suffix (the destination package can be customized by using the `@PStubPackage` annotation in the "master" actor). The stub classes implement the `java.lang.Runnable` interface, and the code of the stub, as it happens in the concrete Java class generated by JavaSPI, is placed inside the `doRun` method. The translation (from the abstract actor

$S($`if(`$bool\text{-}expr$`){`$statement\text{-}list$`}`$)$ ↓ `if(Verify.getBoolean()){` $S(statement\text{-}list)$ `}`
$S($`if(`$bool\text{-}expr$`){`$statement\text{-}list_1$`} else {`$statement\text{-}list_2$`}`$)$ ↓ `if(Verify.getBoolean()){` $S(statement\text{-}list_1)$ `} else {` $S(statement\text{-}list_2)$ `}`
$S($`event(`$name$, $arg\text{-}list$`);` $statement\text{-}list$$)$ ↓ if $name \in \mathcal{R}$ and $name \notin \mathcal{L}$ $evt\text{-}interface\text{-}impl.name(arg\text{-}list)$`;` `evMan.`$name$`_notify(`$arg\text{-}list$`);` $S(statement\text{-}list)$

$S($`event(`$name$, $arg\text{-}list$`);` $statement\text{-}list$$)$
↓ if $name \notin \mathcal{R}$ and $name \in \mathcal{L}$
`if(evMan.`$name$`_isEnabled(`$arg\text{-}list$`)){`
$evt\text{-}interface\text{-}impl.name(arg\text{-}list)$`;`
$S(statement\text{-}list)$
`} else { return; }`

$S($`event(`$name$, $arg\text{-}list$`);` $statement\text{-}list$$)$
↓ if $name \in \mathcal{R}$ and $name \in \mathcal{L}$
`if(evMan.`$name$`_isEnabled(`$arg\text{-}list$`)){`
$evt\text{-}interface\text{-}impl.name(arg\text{-}list)$`;`
`evMan.`$name$`_notify(`$arg\text{-}list$`);`
$S(statement\text{-}list)$
`} else { return; }`

$S($$statement$; $statement\text{-}list$$)$ ↓ $S(statement\text{-}list)$

Table 4.1. Definition of the $S()$ translation function

model to the stub) is described by the $S()$ function formally defined in Table 4.1. Two additional sets have been defined as $\mathcal{R} = \mathcal{R}_{NI} \cup \mathcal{R}_I$ and $\mathcal{L} = \mathcal{L}_{NI} \cup \mathcal{L}_I$. The first rule that matches, from top to bottom, is applied. The JavaSPI model instructions

that do not match any rule are ignored, because they do not alter the sequence of interaction events defined by the correspondence properties.

In Table 4.1, *statement* represents a single JavaSPI instruction, *statement-list* represents a sequence of JavaSPI instructions, and *evt-interface-impl* is the object (provided by the final application) that implements the interface where the event *name* is defined. The object `evMan` is a singleton instance of the class `EventsManager`, shared among all the process stubs. Objects implementing interfaces and arguments must be passed to the stub constructor by the final application (the constructor of the stub is automatically generated in order to accept the correct number and type of arguments). The `Verify.getBoolean()` method is provided by Java Pathfinder libraries. It is used to introduce a non-deterministic boolean choice during the analysis. In this work is used as condition in `if-else` statements. When Java Pathfinder reaches these statements, it analyzes both "true" and "false" branches, one at time, using backtrack when the end of a branch is reached.

4.4.1 Optimization of the generated code

The generation process described in Section 4.4 can produce Java classes for all the possible queries that respect the grammar illustrated in Figure 4.1 and the constraints defined in Section 4.4. As the generated classes must be general enough to cover all the possible combinations, it is possible, in some cases, to slightly optimize the generated code, in order to reduce the complexity of the Java Pathfinder analysis, by reducing the number of states of the application model.

In the current version of JavaSPI, the following two optimizations of the generated code have been implemented.

The first one can be applied in the `EventsManager` class, when the set $\mathcal{R}_I$ is not empty, and consist of using the native Java `int` type instead of the `java.lang.Integer` type, as a counter of event occurrences. However, since the counter is contained inside a `java.util.HashMap`

```
private static final HashMap<List<Message>, java.lang.Integer> EVT-
    NAME = new HashMap<List<Message>, java.lang.Integer>();
```

replacing the `Integer` type with the native `int` type will produce an error in the code. Instead, it is necessary to replace the `Integer` type with an array of `int` type

```
private static final HashMap<List<Message>, int[]> EVT-NAME = new
    HashMap<List<Message>, int[]>();
```

if the array size is 1, the behaviour of the application does not change when compared to the version that uses the `java.lang.Integer` type. In addition to changing the declaration of the `HashMap`, it is necessary to change the implementation of the two methods that operate on the map

```
public synchronized void EVT-NAME_notify(TYPE-PAR-LIST) {
  final List<Message> list= Arrays.asList(PAR-LIST);
```

```
  final int[] v = EVT-NAME.get(list);
  if (v != null) {
    v[0]++;
  } else {
    EVT-NAME.put(list, new int[] { 1 });
  }
}

private boolean EVT-NAME_injDecrease(TYPE-PAR-LIST) {
  final List<Message> list= Arrays.asList(PAR-LIST);
  final int[] v = EVT-NAME.get(list);
  if (v != null && v[0] > 0) {
    v[0]--;
    return true;
  } else {
    return false;
  }
}
```

This optimization dramatically reduces the number of `new` instructions that are called from the application. In fact, since the `Integer` object is immutable, increasing or decreasing the integer value causes the creation of a new `Integer` object (the autoboxing feature of Java hides the process of creating a new object). Instead, if an array of `int` is used, the creation of a new object is executed only once, in the *EVT-NAME*`_notify` method.

The second optimization that has been implemented in JavaSPI, and that involves the `EventsManager` class, consists in not using instances of the `java.util.List` class when events have only one parameter. In fact, the `HashSet` objects track the arguments passed when a method associated to an event (that belongs to $\mathcal{R}_{NI}$) is called. Similarly, the invocations of methods associated to events that belong to $\mathcal{R}_I$ are tracked, and counted, by using `HashMap` objects. By default the events may have an unbounded number of parameters and, as a result, the code generator produces the following code to declare the objects

```
private static final HashSet<List<Message>> EVT-NAME = new HashSet<
    List<Message>>();

private static final HashMap<List<Message>, java.lang.Integer> EVT-
    NAME = new HashMap<List<Message>, java.lang.Integer>();
```

However, it is pretty straightforward to conclude that, if an event has only one parameter, the type maintained by the `HashSet` and the type of the key of the `HashMap` can be changed from `List<Message>` to `Message`. In addition to the declarations, it is necessary to change accordingly the code inside the other methods of the `EventsManager` class.

This optimization can reduce the number of states that have to be analyzed by Java Pathfinder, because it reduces the number of instructions, and object creation

statements, contained in the `EventsManager` class.

Results The statistics of Java Pathfinder show a reduction of the state space, in some simple applications, of about 20% for the first optimization and 15% for the second one. However, these results are very dependent on the complexity of the application under analysis, and optimizations can have less impact in complex applications. In any case, it can be asserted that the two optimizations, when they can be applied, contribute to reduce the state space that has to be explored by Java Pathfinder. The two optimization can be applied at the same time, and the code generator produces a code that is already optimized.

4.4.2 Replace the protocol code with the stub code

The concrete protocol code generated by JavaSPI (Section 2.3) can be either used as standalone application, or imported in a Java application. If called inside an application, the protocol code generated by JavaSPI is used in the following way (the excerpt of code regards one actor of the protocol):

```
...
//object that implement the EventsInterface
InterfName impl = new EvtImplInterf(...);
//object used as argument
Integer val = new IntegerSR(10);
//create a channel object, maybe TCP/IP or software sync
Channel ch = new ...ChannelSR(...);
//class generated by JavaSPI
proc_Callback ps = new proc_Callback(ch, impl);
//send the argument to the process
ch.send(val);
...
```

where `EvtImplInterf` is a class that implements the `InterfName` interface. The interface is automatically generated by JavaSPI, and contains the definition of the methods associated with the process events (Section 4.4).

In order to use the process stub and reduce the complexity of the Java Pathfinder analysis, it is necessary to slightly modify the application code. For example, the previous code must be modified in this way:

```
...
//object that implement the EventsInterface
InterfName impl = new EvtImplInterf(...);
//object used as argument
Integer val = new IntegerSR(10);
//stub class generated by JavaSPI
proc_EventsStub ps = new p_Server_EventsStub(impl,val);
//start a thread with the stub
Thread ts = new Thread(ps);
```

```
ts.start();
ts.join();
...
```

The main differences are: the channel is no longer used, and the thread must be started manually. The channel is not necessary because the argument(s) are directly passed in the method invocation. The thread(s) must be started manually because there may be more than one process stub and, in order to execute a correct Java Pathfinder analysis, all the stubs must be instantiated before launching the threads.

4.5 Proof

This section describes the proof that the generated stub code is a correct abstraction of the concrete protocol code. It is important to remind that the stub replaces the protocol code in order to reduce the complexity of the Java Pathfinder analysis. Since the communication protocol has already been verified using ProVerif, the Java Pathfinder analysis considers only the methods associated with the events defined by the JavaSPI model of the process (see Section 4.3).

The proof aims to ensure that, when the final application is verified with Java Pathfinder and uses the stub, the set that contains all the possible sequences of invocations of the methods associated to the events (defined in the JavaSPI model), is a superset of the set containing all the possible sequences of invocations of the methods associated to the events when the concrete protocol code is used by the final application (motivation is given in the continuation of this section).

More formally, let assume that $\mathcal{E}$ is the set constituted by all the events (parameters are not considered, but each event has a single list of parameters, i.e. overloading is not supported, see Section 4.4) defined in all the actors of the JavaSPI model, and that $\mathcal{S}$ is the infinite set constituted by all the possible sequences (both finite and infinite) that are built by events defined in $\mathcal{E}$. In the sequences, the events can be repeated infinitely many times and appear in any order. However, the model checker Java Pathfinder cannot analyze an infinite state space due to its nature, so it is necessary to define the $\mathcal{S}_b$ set as the finite set derived from truncating all the sequences contained in $\mathcal{S}$ to a defined length `n` (finite and greater than zero), such that each sequence is constituted from a minimum of 0 to a maximum of `n` events of $\mathcal{E}$. All the queries specified in the `@PStubQueries` annotation are saved in the $\mathcal{Q}$ set. Using the rules, described in the continuation of the section, the $\mathcal{S}_{b\mathcal{Q}}$ set is defined as a subset of $\mathcal{S}_b$. The $\mathcal{S}_{b\mathcal{Q}}$ contains only those elements where the sequence of events satisfies all the precedences defined by the queries of $\mathcal{Q}$. At this point, the $\mathcal{S}_{b\mathcal{Q}}$ contains a finite number of sequences, the length of each sequence is between 1 and `n` events, and contains also an empty sequence (the only with length equal to zero). The sequences contained in $\mathcal{S}_{b\mathcal{Q}}$ represent all the sequences of invocations (of methods associated to the events) that Java Pathfinder has to consider during the

verification with the stubs. The maximum length `n` depends on the complexity of the application under analysis and the computational resources available. When the application uses the concrete code, only one of the sequences belonging to $\mathcal{S}_{b\mathcal{Q}}$ is followed for the first `n` invocations of methods associated to the events. However, at each run of the application the sequence followed my change (e.g. due to different inputs, scheduling of processes).

The following steps prove that: (i) the application, when it uses the concrete protocol code, cannot invoke the methods associated to the events in a sequence (for the first `n` steps) that does not belong to $\mathcal{S}_{b\mathcal{Q}}$, (ii) the model checker Java Pathfinder, when it analyses the application (that uses the stubs), considers all the sequences of events that belong to $\mathcal{S}_{b\mathcal{Q}}$ (i.e. a superset of all the possible sequences of methods associated to the events that the final application can follow, for the first `n` steps, are verified). The prerequisites for the validity of this proof are: (i) the ProVerif model generated from the JavaSPI model must satisfy all the security properties defined, (ii) the stubs and the concrete protocol code must be directly generated from the JavaSPI model (the same JavaSPI model mentioned in the previous point).

Define precedences among events: the syntax limitations of the queries that can be defined in the `@PStubQueries` annotation, and how they are parsed, have been described in Section 4.4. All the events used by the annotation constitute four sets: $\mathcal{R}_I$, $\mathcal{L}_I$, $\mathcal{R}_{NI}$ and $\mathcal{L}_{NI}$ (defined in Section 4.4). The same event may belong to different sets. However, due to syntax limitations, if an event belongs to $\mathcal{R}_I \cup \mathcal{L}_I$, it cannot belong to $\mathcal{R}_{NI} \cup \mathcal{L}_{NI}$, and vice versa. The JavaSPI stub generator parses all the queries and reduces them to a set of $fact$ `==>` *hypothesis* queries. For example, if a query is `evtA ==> evtB ==> evtC`, the parser considers the two queries `evtA ==> evtB` and `evtB ==> evtC`. If the original query has only one `==>` symbol, it is not modified by the parser. All the $fact$ `==>` *hypothesis* queries are saved in the $\mathcal{Q}$ set.

The queries in $\mathcal{Q}$ are used to define precedences among the different events. The `==>` defines a precedence between $fact$ and *hypothesis*, as a logic implication. Obviously, the approach is correct if all the original queries have previously been verified with ProVerif. The event that represent the $fact$ can be reached, in a protocol session, if, and only if, the *hypothesis* is true. For example, the `evtD ==> evtE` query implies that the `evtD` event may be reached in a protocol session if, and only if, the `evtE` event has been previously executed at least once, with the same arguments, and not necessarily in the same session. Differently, the `inj:evtF ==> inj:evtG` query means that the `evtF` event may be reached if, and only if, the `evtG` event has been executed with the same arguments. Because the correspondence is injective, an execution of `evtF` must match an execution of `evtG`. In order to emulate this behaviour, the stub generated by JavaSPI counts the occurrences of `evtG` (which belongs to $\mathcal{R}_I$) with the corresponding arguments and, before executing `evtF` (which belongs to $\mathcal{L}_I$), checks if the counter is greater than zero. If it is greater than zero,

the counter is decreased by one and the application method associated with `evtF` is executed. Otherwise, the stub that wants to call `evtF` is interrupted, because the event `evtF` cannot be activated and an invalid state has been reached. If the query does not define an injective correspondence, it is not necessary to use a counter to track the events that compose the hypothesis of the query. For example, in order to enforce the precedence defined by the `evtD ==> evtE` query, the stub generated by JavaSPI saves the calls to the `evtE` event (which belongs to $\mathcal{R}_{NI}$), along with the argument values. The stub, before executing `evtD`, checks if the `evtE` has previously been called with the same arguments. If yes, the application method associated with `evtD` is invoked. Otherwise, the stub that wants to call `evtD` is interrupted (an invalid state has been reached).

Finally, if an event belongs only to $\mathcal{R}_I \cup \mathcal{R}_{NI}$, and does not belong to $\mathcal{L}_I \cup \mathcal{L}_{NI}$, the event has no precedence dependencies and the associated application method can be always invoked when the stub reaches the invocation point.

All these precedence constraints are enforced in the stubs using the translation rules described in Section 4.4.

Compare the instruction flow of the concrete process code and the stub code The concrete protocol code and the stub code are generated following the same approach. A parser analyzes each instruction of the JavaSPI actors model and produces a corresponding instruction in the destination code, also considering the annotations included in the JavaSPI model.

The syntax of the JavaSPI models, along with the generation process of the concrete protocol code is described in [4]. The stub is a "simplified" version of the concrete protocol where only event interactions and branches are considered. In this way, the process stub can reproduce all the possible sequences of invocations of methods associated to events defined in the process.

In JavaSPI, the only statement that creates a branch is the `if-else` statement. However, in the stub it is not possible to use an `if-else` with the same condition, because model variables used in the boolean condition are not known (e.g. keys, identifiers). Java Pathfinder provides a dedicated method for boolean conditions, i.e. `Verify.getBoolean()`. This method represents a non-determinist boolean choice. When used as condition in a `if-else` statement, Java Pathfinder analyzes both the "true" branch and the "false" branch, using backtracking when the end of a branch is reached. The translation rules for this case are reported in the first and second rows of Table 4.1.

Each instruction of the concrete code can raise an exception, due to wrong parameters (e.g. values received from a channel), or due to system errors (e.g. out of memory exception). An occurrence of an exception breaks the flow of the process, and the subsequent instructions, including the invocations of methods associated to events are not executed. This situation is not considered by the stubs, that consider only a complete run of the process. In fact, the break in the process flow

does not create a new branch, because the constructor `try-catch` is not supported by JavaSPI. This implies that an exception cannot continue the execution in an alternative branch where events are defined. Therefore, the stubs can be considered as over-approximations where breaks caused by exception are omitted, but cover all the possible sequences of invocations of methods associated to events.

4.6 The case study application development

The case study is the development of a client-server application that implements a sort of electronic resource access control. In this application, there are a set of clients and one server (which can accept multiple requests at the same time). Each client has an internal counter for credits, and performs requests to the server, asking for additional credits. The initial value of the counter, also called balance of the client, is zero. The server has a fixed initial value for available credits. Each time the server receives a request from a client, it checks if the variable that counts credits is greater that zero, and greater than or equal the amount requested by the client. If this conditions are true, the server decreases the counter by the requested value, and grants the credits with a response message. All clients and the server store internally the same secret key, shared by all trusted components. The secret keys are assumed to be not accessible, both in the clients and in the server.

The security goals that are considered in this case study are: (i) the server cannot grant more credits than the amount available at the beginning, (ii) each client cannot receive more credits than it has requested and, (ii) for each successful request of the client the server must have granted the credits, and the client should have sent the request to the server.

These properties must be satisfied even in the presence of potential active attackers who may intercept/alter/delete messages transmitted between the actors (clients and server), or create new ones, following the definition of attackers of the Dolev-Yao model.

4.6.1 The Development Workflow

The key idea of the proposed development approach (depicted in Figure 4.2) is to divide the application into two distinct parts, to be developed and verified separately: the protocol, and the application logic.

The protocol is developed according to the JavaSPI model-driven methodology. It includes all communication activities and must satisfy some security properties, specified by the developer.

The application logic can be developed in any way, but it must properly interact with the protocol, by starting protocol sessions and reacting to events.

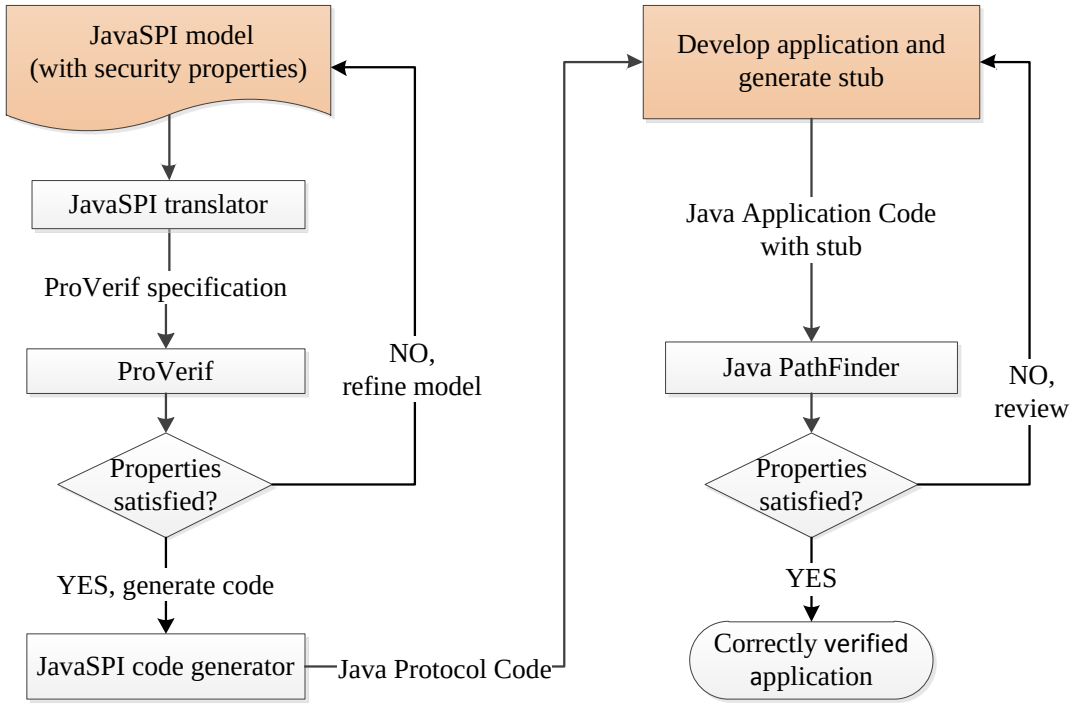

Figure 4.2. Workflow of the verification process

The verification process is compositional. The security properties of the protocol are verified on the abstract protocol model using ProVerif and assuming a generic scenario with an unbounded number of parallel protocol sessions. The same properties are guaranteed to hold on the Java code that implements the protocol by the code generation algorithm. Application-specific security properties are specified and formally verified using an automated formal verification tool capable of analyzing Java code directly (Java Pathfinder in this case). When performing this verification step, it is possible to avoid the explicit modeling of the protocol part, by substituting it with a stub that describes the security properties proved by ProVerif. The stub is be automatically generated from the protocol properties, as described in Section 4.4.

The rest of this section details the various steps with reference to the case study.

4.6.2 Developing the JavaSPI abstract protocol model

The protocol designed for this application is based on challenge interactions. Figure 4.4 shows the interaction between a client and the server during the request operation.

Each client has an unique identifier, named `id`, assigned to the client when it is

launched. The client starts the operation by deciding the amount of credits that will be requested to the server (the `value` term), and calls the method associated to the event `setupRequest`, passing the value as argument. Then, the client generates a random number, called `nonce`. The client compose the request message with the sequence of the client id, the value number and the nonce. This sequence is ciphered with the shared key `sk` (known to clients and server), and sent to the server.

When the server receives a request message, it deciphers the content and passes the value received from the client to the event `checkAvailability`. The implementation of the method that matches the event checks if the server has enough credits to satisfy the client request. If the condition is satisfied, the server replies to the client, otherwise the server closes the connection with the client. The response to the client consist of the hashed value of the sequence received from the client.

Upon receiving the response, the client checks if the hashed value is equal to the expected hash value (calculated with the local values previously sent to the server). If the values match, the client completes the operation and calls the method associated to the event `finishRequest`, passing the value as argument.

In this case study application, the methods associated to the events do the following operations: `setupRequest` counts the total amount of credits requested by each client; `checkAvailability` checks if the server has enough credits to satisfy the client request; `finishRequest` counts the total credits granted by the server to each client.

The JavaSPI specification of the client behavior is the code excerpt shown in Figure 4.3.

The JavaSPI model can be simulated in order to check that it behaves as expected.

This security protocol is expected to satisfy two main security properties. The first one is that the secret shared by all the original components cannot be known by an attacker, who has access to the communication channel between the clients and the server. The second one is the correspondence of the protocol events. In this application, each time a client obtains credits from the server (`finishRequest`), the server must have previously granted the same amount of credits (`checkAvailability`), and the client must have sent the request to the server asking for credits (`setupRequest`). The query that defines the correspondences among events, according to the grammar specified in Figure 4.1, is `inj:finishRequest ==> inj:checkAvailability & inj:setupRequest`. Injectivity is necessary in order to avoid replay attacks (i.e. a duplicated response message, which would result in an addition of not granted credits, must be avoided).

Note that a tool like ProVerif cannot model integer arithmetic and precedence comparisons between integers. Hence, it does not allow to specify more complex properties, e.g. the ones related to the sum of credit granted, nor it allows to describe the application logic that processes the events and updates integer counters.

```
public class Client extends spiProcess {
 ...
 @EventsInterface({"it.polito.javaSPI.eventsExample.gen.ClientInterface
     ","setupRequest", "finishRequest"})
 public void doRun(Channel cClientStart, Channel cClientServer1,
     SharedKey sk) throws SpiWrapperSimException{

 Pair<Integer,Integer> pIdVal = cClientStart.receive(Pair.class);
 Integer id = pIdVal.getLeft();
 Integer val = pIdVal.getRight();

 event("setupRequest",val);

 Nonce n = new Nonce();
 Pair<Pair<Integer,Integer>,Nonce> pIdValN = new Pair<Pair<Integer,
     Integer>,Nonce>(pIdVal,n);

 SharedKeyCiphered<Pair<Pair<Integer,Integer>,Nonce>> skc = new
     SharedKeyCiphered<Pair<Pair<Integer,Integer>,Nonce>>(pIdValN, sk);

 cClientServer1.send(skc);

 Hashing h = new Hashing(pIdValN);
 Hashing rH = cClientServer1.receive(Hashing.class);

 if(rH.equals(h)){
  event("finishRequest",val);
 ...

}
...
```

Figure 4.3. Excerpt of the client model code

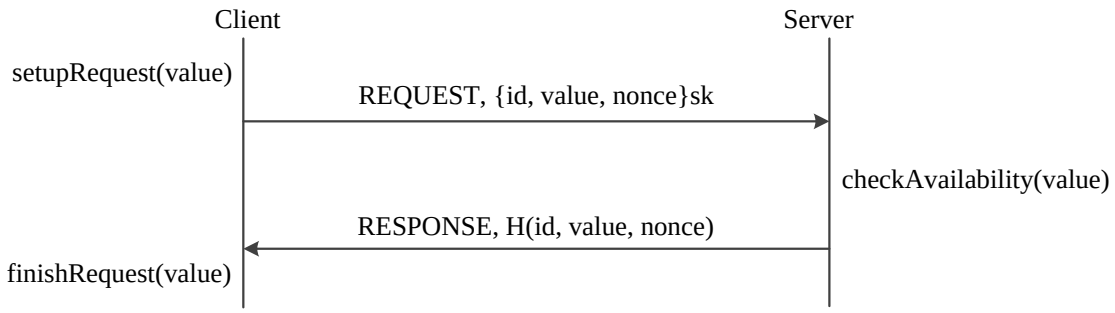

Figure 4.4. The load operation

4.6.3 Formal Protocol Verification

The model generated in the previous step is automatically converted by JavaSPI into the input syntax accepted by ProVerif. The resulting code is ready to be formally analyzed.

ProVerif succeeds in proving that the intended properties of the protocol hold on the model. ProVerif takes 13ms to complete the proof on a computer equipped with Intel i7-3770 CPU running at 3.40GHz, 11GiB of DDR3 RAM, Ubuntu 14.04 64-bit operating system and ProVerif 1.90.

4.6.4 Protocol Code Generation

After having verified the model with ProVerif, the generation of the Java code that implements the protocol can take place, by means of the code generator provided by JavaSPI. The result is a set of Java packages, one for each process in the model, which implements the behavior defined in the model.

4.6.5 Application Logic Development

The generated protocol code must now be integrated with the application code that uses it. In the case study described here, the application code has been kept simple, but it includes all the fundamental aspects of the application that are necessary for its verification. More precisely, only the functionalities related to the management of the credit system have been implemented.

4.6.6 Checking the Application Code

The last step of the workflow is the verification of the application-specific properties using Java Pathfinder. As already anticipated, in order to reduce the complexity of this verification task, the protocol code generated by JavaSPI is replaced with a stub that just reproduces any possible behavior of the protocol sessions, as seen by the application, without really executing the protocol. Of course, the behavior of the stub must be constrained so as to satisfy the security properties that have already been verified by ProVerif. In principle, this constraint can be enforced in one of two different ways: either the constraint is enforced when generating the stub, or the stub is generated without any constraint but the application-specific security property P to be verified is rewritten in the following form

$$C \Rightarrow P$$

where C is the constraint (i.e. the property verified by ProVerif). This second approach is more difficult, because of the difficulty of expressing C. Then, the first

approach (generation of a stub that incorporates the constraints coming from the properties verified by ProVerif has been selected for the case study).

As the application processes interact with each other only through the protocol, having replaced the protocol implementation with the stub makes it possible to avoid considering the behavior of active attackers any more during the verification of the application code. In fact, the behavior of potential active attackers has already been considered when analyzing the protocol by ProVerif, and it is already incorporated in the stub behavior itself.

Based on the architecture of the developed application, the only possible interactions between the protocol and the application logic are those that occur at the start and at the end of each session, as well as at the occurrence of one of the intermediate events described in the model. For this reason, it is enough for the stub to include the statements corresponding to these interaction points. All the other statements that make up the protocol implementation can be safely omitted.

In the case study, the stub includes threads that play the role of the client and threads that play the role of the server (which is a multi-threaded server). The threads that play the role of the client learn the amount of credit to be asked at their startup (this information is an input coming from the user when the application starts the session). Instead, the threads that play the server role are ready to perform the operation.

Model checking does not allow to analyze systems with an unbounded number of states. For this reason, a necessary condition is that the number of parallel protocol sessions (i.e. the number of threads in the stub) is kept bounded. In the case study, this corresponds to having bounded numbers of clients and servers (each operation requires one thread for the client and one thread for the server) and, to limit the number of operations that a client can perform (each client can repeat the request operation more times).

In addition to bounding the number of threads, as with any software model checking problem, abstractions in the application code may be necessary, in order to make the number of states finite and reasonably small.

In the case study, the application-specific properties to be checked are given by the fact that in every instant (or for every state reached and analyzed by the model checker) the value of different integer fields are always inside specific bounds. In details, the counter of available credits (named `available`) in the server, shared and synchronized among different threads that play the role of the server, must always be greater than or equal to zero. This counter is updated by the implementation of the method that matches the event `checkAvailability`. In fact, if the counter is greater than or equal to the value passed to the method as argument, the counter is decreased by that value. Otherwise, the operation is interrupted with an exception. The second condition on integer fields requires that, in each client, the total amount of credits received from the server does not exceeds the total credits requested. This

condition is checked by using two integer fields for each client. The `requested` field is updated by the implementation of the method that matches the event `setupRequest`, by adding the argument value, before sending the request to the server. Similarly, the `balance` field is updated the event `finishRequest`, by adding the argument value, after receiving the response message from the server. The condition that must always be satisfied is that the `requested` field must always be greater or equal than the `balance` field.

Since the instantaneous value of the balance field depends on the field additions and subtractions performed by the application itself, it is not possible to introduce a layer of abstraction on it. Nevertheless, it is still affordable to run the model checker over a reasonable number of possible cases.

To check if it is satisfied there are two possible ways.

The first one is to introduce assertions within the application code. In this case, since the example application requires that integer field respect some conditions, it is sufficient to place an `assert` statement at any point in the code where the fields are modified. As in this application all the fields involved in the analysis are private, it is very simple to identify the only places where then can be set or modified. Specifically, in the server the instruction `assert available >= 0` has been added, while the instruction `assert balance <= requested` has been placed in the client code.

The second way is to use a plugin for Java Pathfinder that enables the verification of LTL formulas during the state exploration performed by JPF. This solution has been described in [5], where a plugin[1], that is not maintained directly by the Java Pathfinder development team, was used. Unfortunately, with the latest update of JPF (year 2015) that plugin does not work anymore with JPF. It was not possible to found another plugin that works with the current version of JPF. However, in the future new plugins might be available in order to extend the functionalities of JPF and enable LTL verification.

Results confirm that the application behaves as expected. No violations of the specified properties are detected, thus proving, by exhaustive state exploration, that the properties hold on the application code.

Verification with JPF was performed on a computer equipped with Intel Xeon Processor E5-2660 CPU running at 2.20GHz and 128GiB of DDR3 RAM. The software components relied on an Ubuntu 10.04 64-bit operating system, Java HotSpot(TM) Server VM (Java version 1.8.0_51-b16, build 25.51-b03, mixed mode).

The initial JavaSPI model is composed by 200 lines of Java code and annotations. The size of the ProVerif model is 110 lines, and the total size of the generated classes (described in Section 4.4) is about 200 lines of Java code. All are generated by the JavaSPI generator starting from the initial model.

Table 4.2 reports elapsed time, required memory and analyzed states for the

[1] Available at https://bitbucket.org/petercipov/jpf-ltl

verification of the case study example. With the computational resources specified above, in this case it has been possible to analyze a scenario with a maximum of 3 clients, and Java Pathfinder can verify up to 3 operations for each client. Java Pathfinder has a limitation of memory that can be used due to the implementation of the state set matcher class (`gov.nasa.jpf.vm.JenkinsStateSet`), which uses an array to store the visited states (the maximum size of a Java array is 2^{31}-1). However, even when Java Pathfinder is not able to complete the exhaustive space exploration, all the states reached before the interruption (an exception is thrown) of the analysis satisfy the specified properties.

Operations \ Clients	1	2	3
1	1s / 1932 MiB / 2013 states	10m 33s / 2847 MiB / 988951 states	70h 43m 4s / 14289 MiB / 322687037 states
2	3s / 1932 MiB / 4761 states	18m 29s / 2762 MiB / 1620845 states	6d 2h 53m 56s / 16383 MiB / 541653845 states
3	4s / 1932 MiB / 6342 states	27m 15s / 3372 MiB / 2380024 states	7d 10h 12m 52s / 31932 MiB / 796832136 states
4	4s / 1932 MiB / 6498 states	27m 27s / 2918 MiB / 2387818 states	exception thrown by Java Pathfinder
5	4s / 1932 MiB / 6654 states	28m 21s / 3373 MiB / 2395612 states	
10	5s / 2436 MiB / 7434 states	32m 37s / 3001 MiB / 2434582 states	
20	5s / 1932 MiB / 8994 states	33m 51s / 3378 MiB / 2512522 states	

Table 4.2. Java Pathfinder verification time, memory consumption and analyzed states using the `assert` construct

Although it is not possible, with a model checker, to formally infer that the properties hold with any number of users and terminals, the results obtained with a small number of participants are sufficient to give reasonable confidence that this is true. In fact, if a distributed application is flawed, usually the error can be detected even with small numbers of parallel sessions.

As mentioned above, the characteristics of the application itself have a significant effect on the complexity of model checking, so performance can be very different depending on the application under test. Finally, some application may reduce the complexity of model checking by exploiting special JPF features, e.g. native peer

classes and customized schedulers. Again, the applicability of those features strongly depends on the application under test.

4.6.7 The `spiWrapperJpf` library

The Java code generated with JavaSPI uses the `spiWrapper` library, which serializes data transmitted over sockets with the default Java object serialization feature. The `spiWrapper` library can be extended in order to achieve interoperability with other software. This library performs many safety checks (e.g. uninitialized objects) that are fundamental when it is used with the concrete communication protocol implementation generated by JavaSPI. However, the complexity of the `spiWrapper` library significantly increases the resources (i.e. memory and time) needed by Java Pathfinder to complete the analysis. To solve this problem, JavaSPI defines a new library, called `spiWrapperJpf`, that redefines the previous `spiWrapper` types with new classes that only implements the data manipulation functions, excluding the codifying layer handling. The application code does not need to be changed: the only modification required is to change the `import` instructions. For example, the case study described above uses the `Integer` type of the `spiWrapper` library for the different counters. The `Integer` type defined by the `spiWrapperJpf` library can be used to replace the former type during the Java Pathfinder analysis. Both classes have the same method signature and extend the `it.polito.spi2java.spiWrapper.Message` class. However, the latter `Integer` type do not implement all the serialization/deserialization function. In fact, these features cannot be invoked by the final application, but only by the generated communication protocol.

Table 4.3 shows the results of the Java Pathfinder analysis of the same case study described above, but where the `spiWrapper` library has been replaced with the `spiWrapperJpf` library.

Operations \ Clients	1	2	3
1	1s / 1932 MiB / 1329 states	2m 20s / 4580 MiB / 493962 states	10h 49m 16s / 7117 MiB / 106535444 states
2	2s / 1932 MiB / 2962 states	3m 33s / 5387 MiB / 712161 states	16h 53m 39s / 5767 MiB / 158434108 states
3	2s / 1932 MiB / 3747 states	4m 55s / 5388 MiB / 971571 states	24h 18m 52s / 10360 MiB / 218779203 states
4	2s / 1932 MiB / 3861 states	4m 59s / 5388 MiB / 976149 states	24h 57m 6s / 10667 MiB / 218933863 states
5	2s / 1932 MiB / 3975 states	4s 59s / 7808 MiB / 980727 states	25h 46m 18s / 9770 MiB / 219088523 states
10	2s / 1932 MiB / 4545 states	5m 6s / 7809 MiB / 1003617 states	29h 51m 12s / 31051 MiB / 219861823 states
20	2s / 1932 MiB / 5685 states	5m 27s / 5390 MiB / 1049397 states	31h 1m 49s / 29203 MiB / 221408423 states

Table 4.3. Java Pathfinder verification time, memory consumption and analyzed states using the `assert` construct and the `spiWrapperJpf` library

Part II

Mobile protocols security analysis

Chapter 5

Formal verification of LTE and UMTS handover procedures

5.1 Introduction

Mobile communication networks are rapidly evolving. The Long Term Evolution (LTE) was standardized in 2008, and it represents the fourth generation (4G) evolution in mobile networks. LTE is an evolution of the previous third generation (3G) Universal Mobile Telecommunications System (UMTS), and nowadays (2015) is already available in in most of the countries of the world. For a considerable period of time these two technologies will co-exist, because the new devices on the market, such as smartphones, at this time support both connection technologies. The older Global System for Mobile Communications (or GSM, second generation) is still used, but the modern mobile terminals prefer to use UMTS and LTE, in order to exploit the higher bandwidth provided by third and fourth generation networks.

Enabling seamless user mobility is a key factor in the LTE and UMTS standards defined by the 3GPP (3^{rd} Generation Partnership Project)[53]. Different procedures have been specified in order to ensure continuity of service to users who move, for example, from an area which is covered by an LTE cell to an area covered by another adjacent LTE cell. Similarly, the standards define procedures to seamlessly move from an area where both 4G and 3G networks are available to an area with only 3G network coverage (or vice versa). In particular, these scenarios where different technologies are cooperating require non-trivial procedures. In fact, an important difference between 3G and 4G networks is that the latter have a flat-IP architecture (all network devices communicate over IP technology), unlike 3G, where communications between devices use âĂŃâĂŃradio channels with multiple access technologies.

In the past, formal verification has already been applied to security protocols for mobile networks. In particular, many works in the literature have formally analyzed the basic procedures for authenticating users in 3G and in 4G networks, while a

smaller number of studies has been devoted to the procedures that allow user mobility in these networks. As a consequence, not all the possible mobility scenarios already have a formal analysis.

The 3GPP defines as IRAT (Inter-Radio Access Technology) handover the procedures in which it is necessary to map the existing security context (ciphering keys, user data) in the transition between two different technologies (such as for example from LTE to UMTS). Instead, the procedures activated when a connection must be seamlessly moved between two LTE network nodes are called Intra-Handover procedures.

Intra-Handover procedures have been formally analyzed in [54], while recently the results of a formal analysis of the IRAT handover procedures that enable users to seamlessly switch from a 3G to a 4G connection, and vice versa, have been presented in [7].

This section of the thesis provides a thorough formal analysis of LTE-LTE and LTE-UMTS procedures, which extends the results previously provided in [54] and in a previous conference paper [7]. In particular, the analysis of LTE-LTE handover procedures includes the verification of some additional aspects that were not considered in [54], and the analysis of LTE-UMTS procedures also considers scenarios including emergency calls. Moreover, a thorough description and motivation of all the formal models used for the analysis and of the underlying design choices is provided.

The tool used for formal analysis is ProVerif (Section 2.1). The security properties that are considered in this work are secrecy of all the keys used before, during and after the handovers, secrecy of payloads exchanged and authentication between network components. In addition to the bare security properties mentioned above, this work also analyses some more specific security properties: backward and forward secrecy of keys, conditional secrecy of payloads (i.e. secrecy that must hold only when optional encryption of data is enabled) and immunity from off-line guessing attacks. The results that have been obtained show that in some particular scenarios, allowed by the standards, and common in real network deployments where IP network security mechanisms are omitted, the aforementioned security properties are only in part assured in the models that have been developed. In these cases, confidentiality of user data traffic is not always provided, and the lack of authentication between network elements makes injection of fake signaling messages possible. This kind of result may be interesting especially for mobile operators, who have to assess security risks in their networks.

5.2 UMTS and LTE overview

This section presents the basic concepts of 3G and 4G mobile networks, which are essential in order to understand the work presented in this thesis. For further details,

refer to the 3GPP specifications [53].

5.2.1 UMTS overview

Figure 5.1a shows the architecture of a UMTS network. The different components are grouped into three domains: the Mobile Station (MS), Serving Network (SN) and Home Network (HN). The mobile station domain is composed of the Mobile Equipment (ME), which is the mobile device, and the Universal Subscriber Identity Module (USIM). The latter contains a worldwide unique identification number, called International Mobile Subscriber Identity (IMSI), and other information shared with the Authentication Center (AuC) of the mobile operator (more details to follow). The Universal Terrestrial Radio Access Network (UTRAN) is the access network for UMTS networks. The UTRAN is composed of Radio Network Controllers (RNCs) and base stations, called NodeB. The RNC is the control unit of the UTRAN network (a single RNC can control a large number of NodeB, which have minimal functionality and mainly propagate messages between MS and RNC). The SN may belong to the same provider of the USIM or to another provider, in areas not covered by the provider of the USIM. The SN is composed of Mobile Switching Centers (MSC) and Visitor Location Registers (VLR). An MSC is able to manage several UTRAN networks. The VLR records information of the MS attached to the network and keeps track of the MS positions. In a real UMTS network, the MSC works in combination with the Serving GPRS Support Node (SGSN) and the Gateway GPRS Support Node (GGSN): the SGSN and the GGSN manage packet data connections (i.e. Internet), while the MSC manages circuit switched connections (i.e. voice). However, in this thesis, they are assumed to be the same entity (i.e. the MSC), because the differences between them are related to technical aspects (i.e. packet data vs. circuit switched connections), but the security aspects are the same for all the devices (i.e. position in the network, security characteristics of interconnections). The home network contains the MSC (the operation is similar to those of the SN), and Home Location Registers (HLR), which contain persistent information on registered operator users, and records the locations of users. Finally, the Authentication Center (AuC) is used to generate the authentication data. For each subscriber identified by the IMSI, it contains the security algorithms and an individual key (K_i) which is a copy of the K_i permanently stored on the USIM card of the subscriber. The IMSI value is public, and can be read from the device that mounts the USIM. The key, however, must remain secret, and must never be revealed by USIM and AuC. For this reason, the USIM provides functions, accessible to the ME, that can be used during the authentication phase in order to obtain temporary keys from K_i. In this way, the secret K_i is never revealed to the ME.

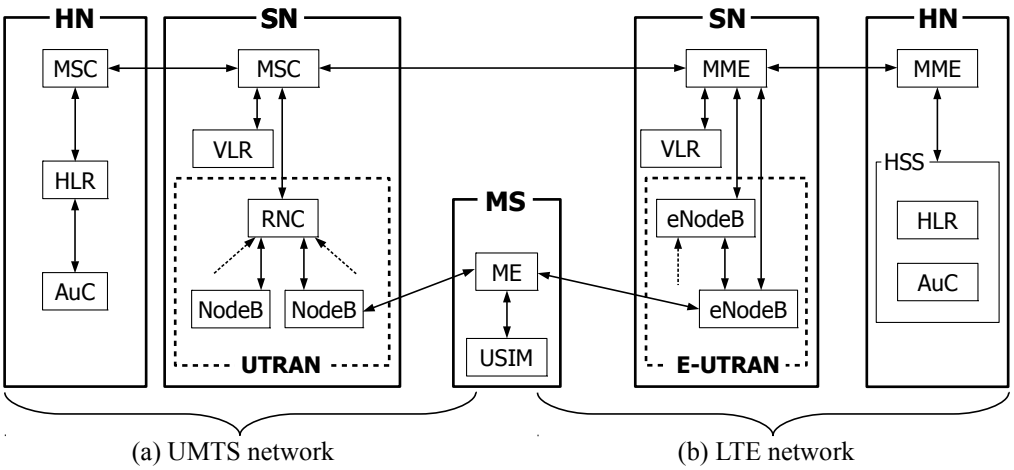

Figure 5.1. UMTS and LTE network architectures

5.2.2 LTE overview

Figure 5.1b depicts the architecture of an LTE network. Unlike the UTRAN, where a RNC controls many NodeB, the Evolved Universal Terrestrial Radio Access Network (E-UTRAN) is composed of only one type of element: the Evolved NodeB (eNodeB or eNB). A Home-eNB (HeNB) performs the same function of an eNodeB, but is optimized for deployment for smaller coverage than macro eNodeB, such as indoor premises and public hotspots. Thus, in this part of the thesis the acronym eNB will be used to refer both to eNodeB an Home-eNB. The eNB are "logically" connected directly to the Mobility Management Entity (MME). In reality, if the eNB-MME connections are protected with IPsec, as 3GPP specification recommends, security gateways are placed between E-UTRAN and MME to terminate IPsec tunnels. However, using IPsec tunnels is at discretion of network operators.

A major difference of the system architecture between LTE and UMTS network is that features that were performed by RNC in the UMTS have now been distributed between eNB and MME. The MME is the main control component for the access network and initiates the authentication process, keeps track of the positions of MS, retrieves subscriptions of MS by HN, and manages connectivity. In LTE, the "concatenation" of HLR and AuC is represented by the Home Subscriber Server (HSS), a single component that combines the functionality of HLR and AuC.

5.2.3 Key hierarchies in LTE and UMTS

Both in LTE and in UMTS, the first procedure done by a mobile device that wants to connect to the network is the Authentication and Key Agreement (AKA) procedure. The objective of this procedure is to establish the keys to be used in cryptographic

operations during communication between mobile device and network. The keys are derived from the shared key K_i and from some randomly generated values. Details of authentication procedures can be found in [53] (TS 33.401). The keys are renewed periodically, in order to prevent possible attacks due to encryption of large volumes of data with the same keys.

The AKA procedure in UMTS networks determines two keys: the Cipher Key (CK) and the Integrity Key (IK), respectively used to encrypt and check the integrity of data exchanged between MS and RNC. UMTS defines only one class of traffic between MS and the network. Thus, only one pair of keys is established (Figure 5.2, right side), which is used for all communications between MS and RNC.

The LTE technology introduces significant differences in key management [53] (TS 33.821). LTE uses different keys for different protocols used between the terminal and the different components of the serving network. These keys are organized in a hierarchy as shown in Figure 5.2 (left side). At the top (root), the key K_i shared between USIM and AuC. The other keys are derived from K_i, following the levels of the hierarchy from top to bottom. Each level of the hierarchy indicates which parts of the network know the keys in the level. As expected, the mobile device knows all the keys except K_i. As in UMTS, starting from the key K_i, the CK and IK keys are derived, even if they are not actually used for encryption and integrity in LTE networks, but rather are used to derive the successive keys. Following the hierarchy, the K_{ASME} key, generated during authentication, is derived by the HSS and then sent to the MME (in the same way, the MS derives the same key). The K_{eNB} key is derived by MS and MME, starting from K_{ASME}, and then sent to the eNB, which can thus activate the security procedures between eNB and MS. However, K_{ASME} and K_{eNB} are not directly used in cryptographic operations. LTE provides two mechanisms of protection for two different classes of control traffic (Control Plane): Non Access Stratum (NAS) traffic, and Access Stratum (AS) traffic. NAS traffic consist of communications between MME and MS (forwarded in a "transparent" way through the eNB), while AS traffic (also called Radio Resource Control (RRC) traffic) includes the control messages between MS and eNB. For this reason, two keys are derived from K_{ASME}: K_{NASenc}, used for encryption, and K_{NASint}, used for integrity checking of NAS messages. Similarly, from K_{eNB}, the keys K_{RRCenc} and K_{RRCint} are derived and used for AS messages. The user traffic (User Plane), is encrypted using a different key, called K_{UPenc}. Integrity protection is not supported for this class of traffic.

Finally, after a successful handover of the MS between two neighbor eNB, it is necessary to renew the K_{eNB} [53] (TS 33.401). To do this, the MME derives a new value from the key K_{ASME}, called Next Hop key, which is used, along with the previous K_{eNB}, to generate the K_{eNB} key (called $K^{\star}_{eNB}$) used by the target eNB after the handover. Further details on these procedures and their analysis can be found in [53] (TS 23.401 and TS 33.401) and [54] respectively.

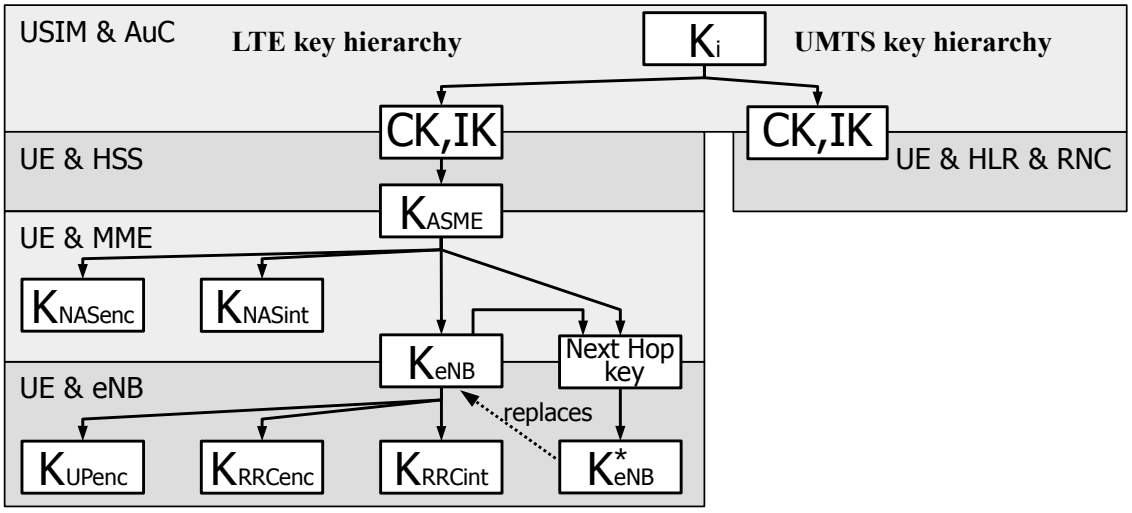

Figure 5.2. LTE and UMTS key hierarchies

5.2.4 Handover procedures

Handover procedures are activated by the serving network (eNB in LTE, RNC in UMTS) when the strength of the radio signal between a mobile station and the current eNodeB/NodeB becomes too much degraded. The decision of performing a handover is taken by the eNB or RNC, which selects the target eNB/RNC from a list of neighbors (the list is previously known). When a neighbor with the same technology (LTE/UMTS) is not available for the handover, then a handover to a network with other technology is executed. Intra-Handover procedures are adopted when a user moves between different LTE cells, while Inter-RAT procedures are adopted while moving from a radio access technology (GSM, UMTS, LTE, WiMAX or any other wireless technology) to another. These procedures are described in the 3GPP TS 23.401 and TS 33.401 specifications [53]. eNBs can be directly connected by an X2 interface which can be used to perform handover procedures. Instead, eNBs are connected to the MME via the S1 interface. Both interfaces are IP based.

5.3 Security requirements and threats

The handover procedures have different security requirements, as specified by the 3GPP standards. All the procedures, assuming that the mobile device is authenticated with the network components (MSC in UMTS, eNB and MME in LTE) before the handover begins, must guarantee the validity of the same authentication properties after the handover is completed, in the destination network. Similarly, all the procedures must keep the secrecy of all the keys used before, during and after the handover in the mobile device and in the operator network. Consequently, the

procedures for handover always activate the protection of the data transmitted with the exception for unauthenticated emergency call when integrity checks and ciphering procedures cannot be applied.

Security threats derive from different causes. While physical damages and technical failures are out of the scope of this work, the analysis considers malicious threats originated by attackers who can eavesdrop, alter and drop communications between the mobile device and the operator network, and among some components of the operator network, even when emergency calls are ongoing. In this scenario, the threat consequences, in the handover procedures, may be the disruption of authentication between components and loss of data privacy.

In order to counter security threats, communication among components of the home and serving network should be secured by the mobile operators that own the networks. While the risk of attacks on the MSC-MSC, MME-MME, MME-MSC and MSC-RNC links is not very relevant, because the involved nodes are not physically accessible, the same is not true for the eNB-MME and eNB-eNB links, especially in the case of HeNBs, because these nodes are often located in publicly accessible locations, and hence they may be tampered by a malicious attacker. The 3GPP TS 33.820 and 33.401 [53] specifications specify that the eNB-MME and eNB-eNB connections should be protected by IPsec, which guarantees authentication, integrity and confidentiality of data. Moreover, Security Gateways (SeGW) should be used to handle the IPsec connections in the serving network. However, the 3GPP TS 33.401 [53] specification reports that, if the interfaces are trusted (e.g. physically protected), the use of IPsec based protection is not needed, depending on operator evaluations. In practice, several operators avoid using IPsec on their networks. Reasons might be several: some fear that IPsec would increase both network complexity and traffic latency, others simply underestimate the problem as, for example, they assume that encryption is performed by applications, which is not always true. A clear presentation of all the possible motivations that are leading several network operators to avoid using IPsec is available in [55].

5.4 Modeling handover procedures for security verification

5.4.1 Modeling choices

This section presents the main modeling choices made in developing the formal models of the handover procedures. The final aim is to create models that faithfully represent the procedures to be analyzed but that are as simple as possible, so as to efficiently exploit the analysis tool ProVerif.

Omitting non-relevant data and operations

When modeling handover procedures for analyzing their security, only the data and operations related to cryptography and authentication need to be included in the models, while information related to resource allocation and relocation is not relevant for the security analysis and can be omitted.

Abstracting algorithms and algorithm identifiers in key derivation functions

Since perfect cryptography is assumed in the Dolev-Yao attacker model, the handover models consider only whether encryption is enabled or not, no matter which algorithm is chosen. Therefore, the algorithms and the algorithm identifiers are abstracted away from key derivation functions.

Using a single fresh value to represent an IMSI

An IMSI consists of three parts [53] (TS 23.003): (i) Mobile Country Code (MCC), which identifies the country that the subscriber domiciles, (ii) Mobile Network Code (MNC), which identifies the HN of the subscriber, and (iii) Mobile Subscriber Identification Number (MSIN), which identifies the subscriber within the HN. As the splitting of an IMSI into its components is not relevant for this analysis, in this work a single value is used to represent the IMSI. As subscribers are uniquely identified by their IMSI, an IMSI is modeled as a fresh value, i.e. as a value generated before the start of the protocol and guaranteed to be unique. Fresh values are considered by ProVerif initially unknown and unguessable by the attacker, while in practice an active attacker can obtain a subscriber's IMSI using so-called IMSI catchers. In order to take this into account, in the models the MS sends its IMSI in clear over the public channel in the first message. Thus, the attacker can learn the IMSI by eavesdropping on the public channel.

Modeling AKA procedures

As the handover procedures can be activated at any time, when the MS is already authenticated with the serving network, and the previous authentication state is important, the model cannot just include the procedures themselves, but it needs to represent what may happen before the procedures are activated. Most notably, the model should include the last AKA procedure that has been executed by the entities involved in the handover. As the inclusion of a full AKA procedure model would

make the overall model too complex to be analyzed[1], the initial authentication is not fully modeled, but it is substituted by an equivalent model, which creates the same security context that is assumed to be established by the executed AKA procedure. This modeling choice was also adopted in [54].

In each AKA equivalent model, a fresh term used as IMSI is first generated by the MS, and whether to activate encryption or not is non-deterministically chosen, so as to consider both cases.

In the LTE to UMTS, LTE X2 and LTE S1 handover models each MS also generates a fresh term used as K_{ASME} (that in reality is established during the AKA). Encryption selection and K_{ASME} are inserted as values in private perfect hash tables, shared only with the MME and called `capab` and `keys`. In these tables, the corresponding IMSI is used as key for selecting the corresponding values. So, the MME can retrieve the correct values for each MS from these hash tables, by using the IMSI value (which is public). In other words, the agreement achieved by the initial AKA context setup is replaced by the two shared tables. Such tables, being private, cannot be accessed by the attacker. Here are the ProVerif code segments that represent the handling of the shared data:

```
(* define two tables *)
table keys(ident , asmeKey).
table capab(ident , bool).

(* generate fresh IMSI *)
new imsi: ident;
(* nondeterministically chose a value between true and false *)
let cap_ue: bool suchthat mem(cap_ue , uecaps) in
(* generate a fresh term used as KASME *)
new kasme: asmeKey;
(* insert new terms into the tables *)
insert capab(imsi, cap);
insert keys(imsi, kasme);

(* retrieve terms from the tables, using IMSI as key *)
get keys(=imsi, kasme_recv) in
get capab(=imsi, cap_recv) in
```

Instead, in the UMTS to LTE models, in addition to nondeterministically selecting whether encryption is enabled or not, the MS also generates two fresh terms used as ciphering and integrity keys in UMTS (CK,IK), that in reality are established during the AKA. Similarly to the previous case, the selected encryption capability and the (CK,IK) key pair are inserted as values in private perfect hash tables, shared only with the MSC, called `capab` and `keys`. The corresponding IMSI value (which is public) is used as key for addressing these tables, thus allowing the MSC to retrieve the correct values for each MS.

[1]The inclusion of the complete model of AKA procedure caused the non-termination of the ProVerif analysis.

Modeling communication channels

Communication channels are modeled according to the considerations made in Section 5.3. Given that the MSC-MSC, MME-MME, MME-MSC and MSC-RNC links are generally not physically accessible to attackers, while the eNB-MME and eNB-eNB links are often accessible and not endowed with IPsec protection, in the analysis it is assumed that the MSC-MSC, MME-MME, MME-MSC and MSC-RNC links are secure channels, i.e. not accessible by the attacker, whereas for the eNB-MME and eNB-eNB links this thesis explores both the case that the channels are secured by IPsec, and hence actually not accessible by the attacker, and the case that an attacker may be able to control the channels, which is a possibility if the operator does not use IPsec protection for these channels and the attacker succeeds in having access to a trusted interface.

One simple possible way of modeling a secure channel in ProVerif is to use a private channel, which, by definition, cannot be accessed by the attacker. A second possible way is by encrypting the data that flow through the channel with secret keys that are shared by the end-points of the channel, are not known to the attacker, and are never disclosed. With this solution, the impossibility for the attacker to access the secure channel is guaranteed by the Dolev-Yao attacker model which assumes perfect cryptography. The latter method is more complex than the one using a private channel. For this reason, the ProVerif models used in this work adopt the former approach:

```
1  free pubChannel: channel. (* public channel used to connect MS and eNB/RNC *)
2  free secureChannelEnbMme: channel [private]. (* private channel *)
```

Modeling message headers

Each message has a header that identifies the type of message content. Headers are defined as constants in the model. Each process that receives a message checks if the message header matches the one expected for the current input instruction. If it does not match, the message is immediately discarded by the process:

```
1  const HO_REQUIRED: msgHdr. (* message header definition *)
2  in(=HO_REQUIRED, ...)      (* message input with header filter *)
```

This solution faithfully represents the way input messages have to be checked but at the same time it keeps a low footprint on the state space size of the model.

Modeling capabilities

As in Dolev-Yao models the details about ciphering algorithms are omitted, the same is done here: the model only represents whether the MS activates encryption (`true` value) or not (`false` value), but it does not represent other choices (e.g. encryption algorithm). Note that encryption is optional, but integrity protection is mandatory

in LTE. Hence, only the encryption capability has to be represented. As said, the boolean value of this capability is nondeterministically chosen by the MS, so that the analysis considers both cases. The selected value of the capability is disclosed to the attacker in the first message sent by the MS.

```
(* create a set containing only true and false values *)
let uecaps = consset (true, consset (false, emptyset)) in

(* nondeterministically chose a value in the set *)
let cap_ue: bool suchthat mem(cap_ue , uecaps) in
```

Omitting temporary identifiers

In the model, the IMSI is used to identify the MS, while in reality temporary identifiers are used, i.e. Temporary Mobile Subscriber Identity (TMSI) in UMTS, and Globally Unique Temporary Identifier (GUTI) in LTE. This abstraction does not alter the security properties of the procedures, because the attacker can obtain the IMSI from temporary identifiers, as demonstrated by Arapinis et al. [56].

Representing data message exchanges

Before and after the handover procedures take place, data messages can be exchanged. This is taken into account, but only the exchange of two data messages is included, one before the procedure starts and one after its completion, because exchanging more messages would not add anything significant to the model. These messages are also used to check the secrecy of the data traffic when encryption is enabled.

Using a fresh term to model a counter

The LTE to UMTS handover uses a counter to derive the UMTS CK′ and IK′ keys. This counter is called NAS downlink count, and represents the NAS protocol message counter. The counter is bounded, and when it is about to wrap around a new AKA procedure is activated, in order to generate a new set of keys (K_{ASME}, K_{eNB} and all derived keys). Integer values are not directly supported by ProVerif. The increment of the NAS downlink count value is therefore modeled as the creation of a fresh new value, which is disclosed to the attacker in the next message, as shown in the following ProVerif code:

```
new nasDownlinkCount: bitstring;
let ck': ckKey = kdf_ck'(kasme,nasDownlinkCount) in
let ik': ikKey = kdf_ik'(kasme,nasDownlinkCount) in
out(pubChannel,nasDownlinkCount);
```

The disclosure operation models the fact that a counter can be eventually guessed by an attacker, because it is a bounded integer value. Using a private fresh term does not correctly represent a counter in the model, because a fresh term is unguessable. Disclosing the fresh term used as counter is an acceptable approximation because

it adds the counter value to the attacker knowledge database, and covers the case when the attacker guesses the counter value. This design choice was already adopted in [54].

Simplifying transmission paths

In order to reduce the complexity of the analysis, some messages in the models do not follow the real path from source to target, but they follow a simplified path. For example, the HANDOVER COMMAND message (in the LTE to UMTS and UMTS to LTE procedures) is directly exchanged between MS and MME in the model. In reality, this message passes through the eNB node, but the eNB does not alter the contents of the message, unless some physical parameters, and the ciphering and integrity checking, done with the K_{RRCenc} and K_{RRCint} keys. Modeling the path through the eNB, with the additional ciphering and integrity checking, is possible, but leads to models that cause the inability of ProVerif to terminate successfully. This problem has been avoided by introducing a public direct channel between MS and MME, which replaces the sequence of MS-eNB (public channel) and eNB-MME (private channel if protected with IPSec, public otherwise) channels that in reality exist in the network. This replacement is a sound approximation of reality, because it enlarges the possible attacks on the protocol (the MS-MME channel is public, and the ciphering and integrity checking done with the K_{RRCenc} and K_{RRCint} keys is omitted). Hence, if a security property holds on this model, it must hold a fortiori when the real channels are used. Note that the encryption of messages between MS and MME with the K_{NASenc} key is still modeled, when required.

Modeling emergency sessions

LTE redefines the management of emergency calls. Emergency services are handled by the IP Multimedia Subsystem [53] (TS 23.167), and can be activated even if the user is not authenticated (i.e. the MS does not mount a USIM card). During emergency calls, a handover from LTE to UMTS can be performed if necessary, while the handover from UMTS to LTE is not supported [53] (TS 23.401). The ProVerif models of the LTE to UMTS handover consider the possibility that a user may activate emergency mode, in order to verify if an attacker can exploit data acquired during the emergency session handovers to break the security of legitimate communications. Similarly, the models of LTE to LTE handovers consider emergency sessions. Emergency session have been modeled as separate processes, one for each actor, where encryption and integrity checks are disabled. The same IMSI is used to start a MS process that models an emergency terminal (unauthenticated), and one process that follows the authenticated session. By adopting this approach, the models consider the possibility that the same IMSI is used at the same time for an authenticated session and for an emergency session. This possibility in reality

may happen if an attacker uses the IMSI to start an emergency session, while the legitimate user is connected to the network.

5.4.2 Procedure models

The next subsections give an informal description of the procedure models used for security verification, in the form of charts. The models have been derived from the procedure descriptions given in 3GPP TS 23.401 and TS 33.401 [53] specifications, but omitting non security relevant data and operations and following the design choices detailed above.

The equivalent model that substitutes the AKA procedures is inserted at the beginning of each handover procedure model, in order to represent the establishment of the security context assumed before starting the handover procedure itself. Just after the first two messages representing the initial AKA equivalent model, a third message exchange is inserted before starting each handover procedure itself. This message represents a user data exchange between MS and eNB/MME/RNC, done before the handover procedure itself. These initial messages can be seen, for example, in the chart in Figure 5.4, which represents the messages exchanged during a LTE to UMTS handover.

Figure 5.3 contains an excerpt of the ProVerif model used to verify the LTE to UMTS handover procedure, and follows the standard patterns used in ProVerif modeling. The LTE to UMTS handover model is used in this section as reference for describing how the security properties have been verified. All the handover models follow the same modeling technique. The compete handover models are available for download at the URL `http://staff.polito.it/riccardo.sisto/lte.umts.handover/fullmodels.zip`

```
free pubChannel: channel. free secureChannelEnbMme: channel [private].
const HO_REQUIRED:msgHdr. const FWD_RELOC_REQ:msgHdr. const ID:msgHdr.
table keys(ident , asmeKey). table capab(ident , bool).
query attacker(new kasme_ue).
query attacker(kdf_enb(new kasme_ue)).
query x1: ident, x2: enbKey;
 inj-event(endMS_ENB(x1, x2)) ==> inj-event(begMS_ENB(x1, x2)).
query x1: ident, x2: ckKey, x3: ikKey;
 inj-event(endRNC_MS(x1, x2, x3)) ==> inj-event(begRNC_MS(x1, x2, x3)).
query x1: ident, x2: asmeKey, x3: bitstring, x4: ckKey, x5: ikKey;
 inj-event(endMME_MS(x1, x2, x3, x4, x5))
                    ==> inj-event(begMME_MS(x1, x2, x3, x4, x5)).
query attacker(payloadLTE) ==> event(disableEnc).
query attacker(payloadUMTS) ==> event(disableEnc).
...
weaksecret payloadLTE.
weaksecret payloadUMTS.
...
let processMS(uecaps:bset) =
 new imsi_ue: ident;
 let cap_ue: bool suchthat mem(cap_ue , uecaps) in
 new kasme_ue: asmeKey;
 insert capab(imsi_ue, cap_ue); insert keys(imsi_ue, kasme_ue);
 (* key derivation from Kasme *)
 let knasenc_ue: nasEncKey = kdf_nas_enc(kasme_ue) in
 let knasint_ue: nasIntKey = kdf_nas_int(kasme_ue) in
 let kenb_ue: enbKey = kdf_enb(kasme_ue) in
 ...
 event begMS_ENB(imsi_ue, kenb_ue);
 out(pubChannel, (ID, imsi_ue, cap_ue));
 if cap_ue = true then ( (* encryption enabled inside this branch *)
  ...
 )else(
  if cap_ue = false then ( (* encryption disabled inside this branch *)
   event disableEnc;
   ...
  )else ( 0 )
 ).
...
let processMME =
 in(pubChannel, (=ID, imsi_mme: ident, cap_mme_recv: bool));
 get keys(=imsi_mme, kasme_mme) in ( get capab(=imsi_mme, cap_mme) in
  let knasenc_mme: nasEncKey = kdf_nas_enc(kasme_mme) in
  new nasDownlinkCount: bitstring;
  let ck'_mme: ckKey = kdf_ck'(kasme_mme,nasDownlinkCount) in
  let ik'_mme: ikKey = kdf_ik'(kasme_mme,nasDownlinkCount) in
  ...
 ).
process
 let uecaps = consset (true, consset (false, emptyset)) in
 ((!processMS(uecaps)) | (!processENB) | (!processMME) |
  (!processMSC) | (!processRNC))
```

Figure 5.3. An excerpt of the LTE to UMTS handover

LTE to UMTS

Figure 5.4 depicts the simplified message exchange flow performed during Inter-RAT handover from LTE to UMTS technologies, and represents the ProVerif model used for the verification of the handover procedure.

After the first three context messages already explained, the handover is activated by the eNB with the HANDOVER REQUIRED message, which informs the MME that the procedure must be performed for the user identified by the *IMSI* contained in the message. The MME derives the new CK′ and IK′ UMTS keys from the previous K_{ASME} and the *NAS downlink count* value. The FORWARD RELOCATION REQUEST message provides the target MSC with the two keys and the *IMSI*. The MSC provides the target RNC with the keys just received and the user identity (RELOCATION REQUEST message). Now the RNC has all information required to communicate with the MS. RELOCATION REQUEST ACK and FORWARD RELOCATION RESPONSE messages are used to inform that the target UMTS network is ready to accept the connection from MS. The HANDOVER COMMAND is a NAS message that provides the MS with the data (*NAS downlink count*) required for the derivation of CK′ and IK′ in the MS. Then the MS sends a HANDOVER TO UTRAN COMPLETE message to the target RNC for signalling that the MS is ready to use the UMTS network. Finally, two messages are used to establish agreement upon the encryption algorithm, using the SMC (SECURITY MODE COMMAND) and the SMC COMPLETE messages. The last message represents data exchange after the handover, as already discussed.

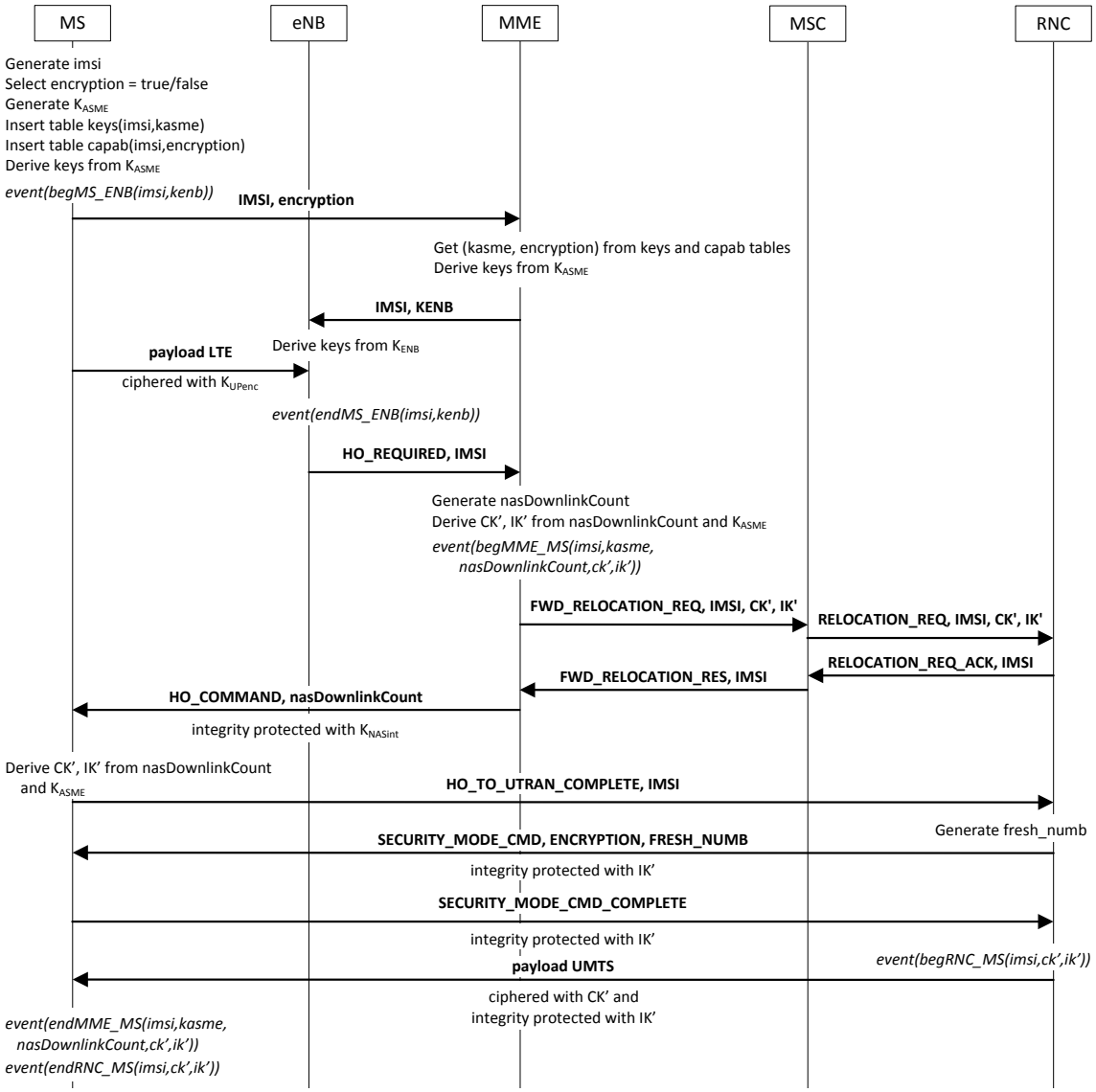

Figure 5.4. LTE to UMTS handover

UMTS to LTE

Handover from UMTS to LTE (Figure 5.5) is similar to the LTE to UMTS handover, but with the network roles reversed.

Handover is activated by the RNC with the RELOCATION REQUIRED message, which informs the MSC that the procedure must be performed for the user identified by the IMSI contained in the message. The MSC forwards the data received from the MSC to the target MME, using the FORWARD RELOCATION REQUEST. The MME computes the new LTE keys following these steps: (i) generates a fresh nonce, (ii) uses a derivation function to obtain a K'_{ASME} key from the nonce, CK and IK received from MSC, (iii) derives the new K_{eNB}, K_{NASenc} and K_{NASint} keys from K'_{ASME}. The K_{eNB} is sent, along with the IMSI and the nonce, to the target eNB (HANDOVER REQUEST message), which confirms the reception with the HANDOVER REQUEST ACKNOWLEDGE message. The eNB can therefore derive the K_{RRCenc}, K_{RRCint} and K_{UPenc} keys from the received K_{eNB}. Then, the MME sends the FORWARD RELOCATION RESPONSE to the MSC, which forwards the nonce to the MS with the HANDOVER COMMAND. Now the MS can derive the complete set of LTE keys from the received nonce and the previous CK and IK. When the derivation process is completed, the MS informs the target eNB with the HANDOVER TO E-UTRAN COMPLETE message. The next four messages activate the security (i.e. agree upon the security algorithms) of the Access Stratum and Non Access Stratum security, respectively between MS and eNB, and between MS and MME. The messages HANDOVER NOTIFY, FORWARD RELOCATION COMPLETE and FORWARD RELOCATION COMPLETE ACKNOWLEDGE completes the handover procedure by signalling to the MSC that the handover completed successfully. Finally, the last message represents data exchange after the handover.

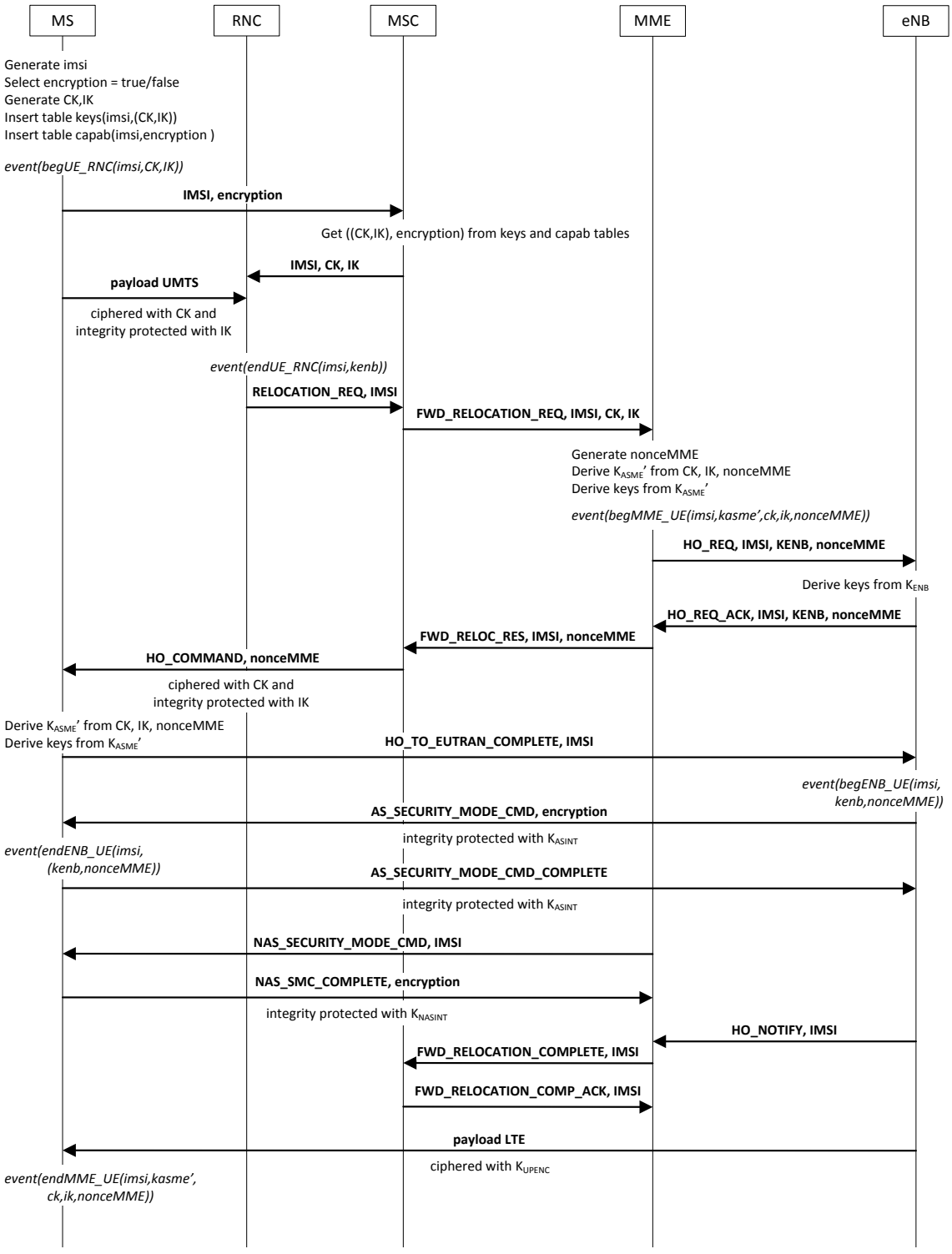

Figure 5.5. UMTS to LTE handover

LTE X2

The X2 handover (Figure 5.6) is an LTE to LTE handover procedure. The fundamental characteristic of the X2 procedure is the fact that the handover is performed between two eNB, without MME intervention. Indeed, the MME is informed that the handover has been performed after the procedure completed. An X2 handover can be executed between two eNB only if they are directly connected via the X2 interface. Otherwise, an S1 handover must be performed (Section 5.4.2).

The X2 handover is initiated by the SeNB (Source eNodeB) deriving the $K^{\star}_{eNB}$ key from the current K_{eNB} and the Target Cell ID, an identifier that is associated by the SeNB to the TeNB (Target eNodeB). The Target Cell ID is modeled as a fresh term that is disclosed to the attacker, because this ID is known by any MS that connects to the eNB, thus the attacker can obtain it by starting a legitimate connection to the eNB. The SeNB informs the TeNB that the handover is starting, by sending $K^{\star}_{eNB}$, MS identity and encryption capability in the HANDOVER REQUEST message.

The TeNB derives the new set of keys (K_{RRCenc}, K_{RRCint} and K_{UPenc}) from the received $K^{\star}_{eNB}$, and informs the SeNB that it is ready to accept the connection from MS (HANDOVER REQUEST ACKNOWLEDGE message). Then, the SeNB sends all the information required (encryption capability, that the MS checks to be corresponding to the value selected at the beginning, and Target Cell ID) to the MS in a RRC CONNECTION RECONFIGURATION message. Now the MS can derive the new $K^{\star}_{eNB}$ key and all the subsequent keys (K_{RRCenc}, K_{RRCint} and K_{UPenc}) that are used to communicate with the TeNB. Thus, the MS disconnects from the SeNB and sends a RRC CONNECTION RECONFIGURATION COMPLETE message to the TeNB. When the TeNB receives this message, it can start to communicate with the MS. Then, the TeNB informs the MME that an X2 handover has been performed with the PATH SWITCH REQUEST. The MME derives two new keys, called next hop key 1 (from K_{eNB} and K_{ASME}) and next hop key 2 (from next hop key 1 and K_{ASME}). The next hop key 2 is sent to the TeNB in the PATH SWITCH REQUEST ACKNOWLEDGE message, and must be used by the TeNB to derive another $K^{\star}_{eNB}$ for the next handover. This implies a two-step forward key separation, because even though the SeNB can derive the key used for the TeNB, it cannot derive a key for the next target eNB. Finally, the last message represents data exchange after the handover.

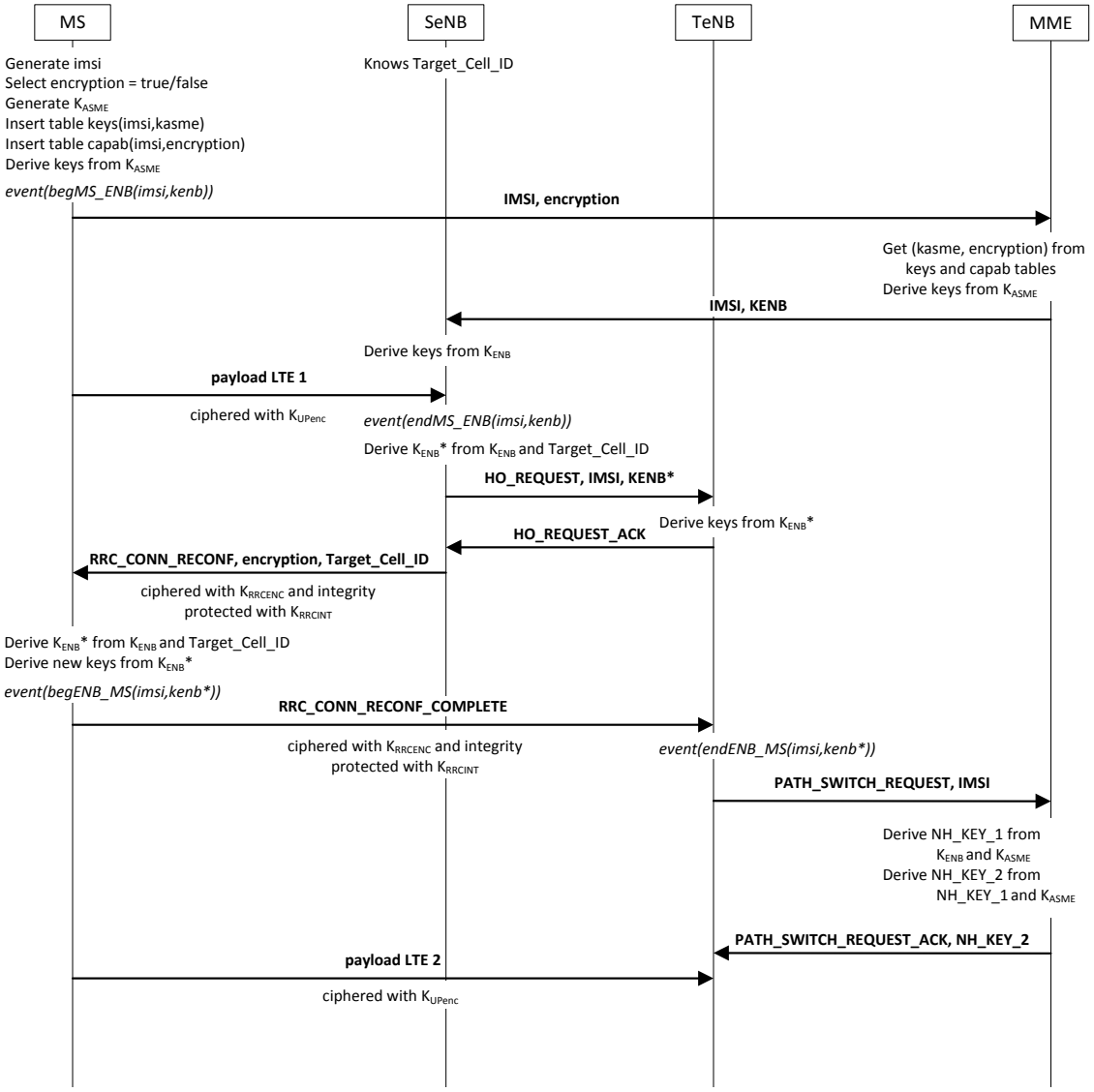

Figure 5.6. LTE X2 handover

LTE S1

The S1 handover (Figure 5.7) is an LTE to LTE handover procedure. Differently from the X2 handover (Section 5.4.2), the S1 handover procedure requires the intervention of the MME.

The S1 handover is initiated by the SeNB deriving the $K^{\star}_{eNB}$ key from the current K_{eNB} and the Target Cell ID (an identifier that is associated by the SeNB to the TeNB, modeled as a fresh term that is disclosed to the attacker). The SeNB informs the MME of the necessity that a handover is required, by sending $K^{\star}_{eNB}$, MS identity and encryption capability in the HANDOVER REQUIRED message.

The MME derives two new keys, called next hop key 1 (from K_{eNB} and K_{ASME}) and next hop key 2 (from next hop key 1 and K_{ASME}). The next hop key 2 is sent to the TeNB, along with the MS identity (IMSI) in the HANDOVER REQUEST message. The TeNB derives the new $K^{\star}_{eNB}$ key from the received next hop key 2 and the Target Cell ID, which is known from the beginning for simplicity. Then, the TeNB derives the following K_{RRCenc}, K_{RRCint} and K_{UPenc} keys. Meanwhile, the MME sends the HANDOVER COMMAND message to the SeNB, which forwards to the MS the message along with the encryption capability (that the MS checks to be equal to the value selected at the beginning) and the Target Cell ID.

The MS can derive the new set of keys: the K_{RRCenc}, K_{RRCint} and K_{UPenc} keys will be used to communicate with the TeNB. Then, the MS disconnects from SeNB and initiates the message exchange with the TeNB by sending the HANDOVER CONFIRM message.

Finally, the last message represents data exchange after the handover.

The S1 handover procedure implies a one-step forward key separation: the SeNB cannot derive the key used in TeNB when the handover is completed, because the keyring material of the TeNB is provided directly by the MME.

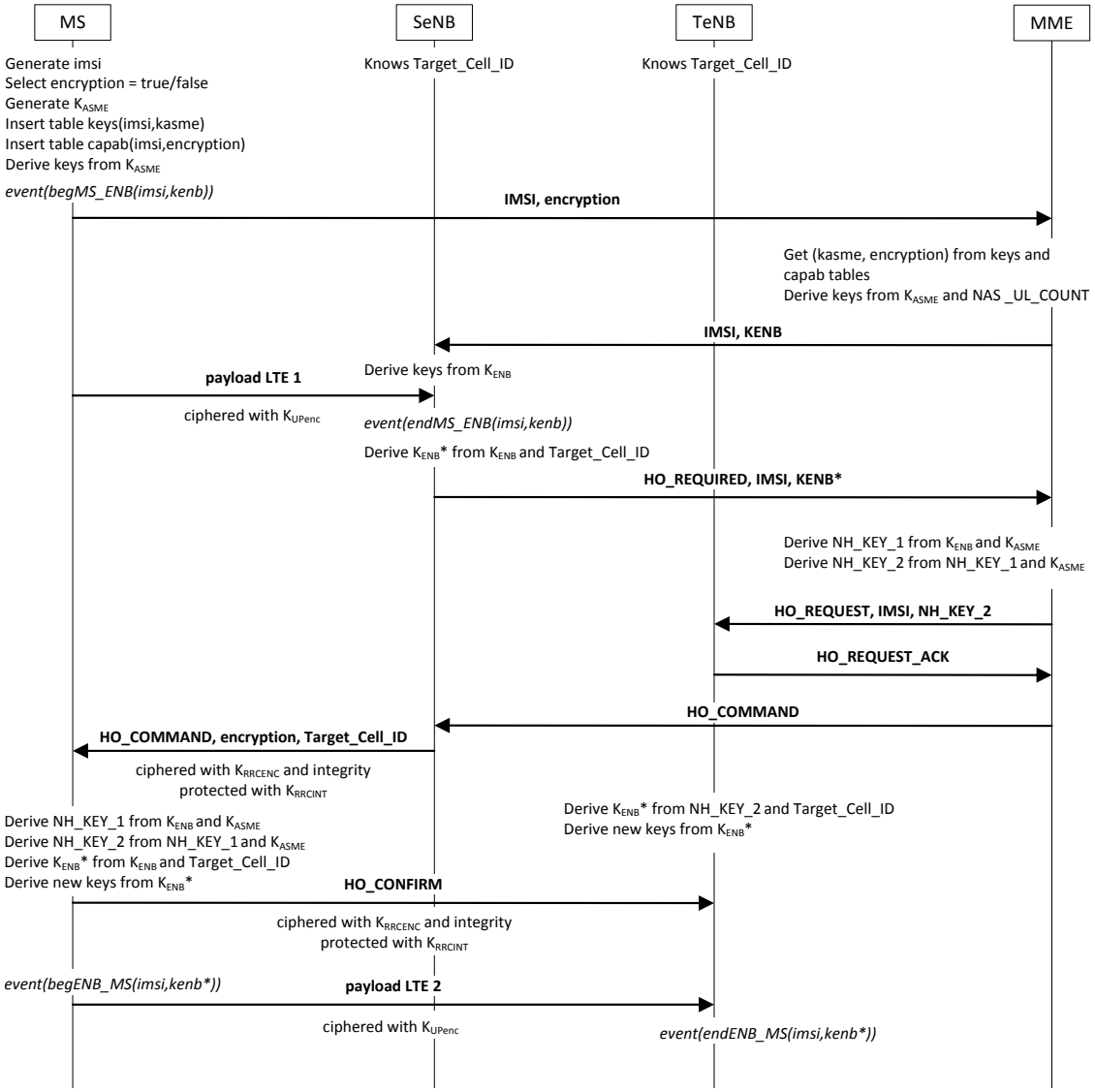

Figure 5.7. LTE S1 handover

5.4.3 Security properties specification

The main security properties that the handover procedures are expected to guarantee have been specified as follows (line numbers refer to the LTE to UMTS ProVerif code in Figure 5.3):

- *Secrecy of keys*: all the keys involved in the handover procedures must remain secret (lines 4, 5).

- *Conditional secrecy of payloads*: in UMTS and LTE, encryption of data between MS and SN is optional, unless an emergency call without authentication is running, in which case encryption is disabled. Accordingly, the terms `payloadLTE` and `payloadUMTS`, used to represent the data transferred between MS and eNB/RNC (when an emergency session is not active), must be kept secret if encryption is enabled. An equivalent formulation is that, if the attacker knows the secret payload, then the event `disableEnc` must have been previously executed (lines 13, 14). Note that the secret payload referred by this property is not the payload of emergency sessions messages, which is represented in the model by another term and is not protected (the attacker can read and modify it).

- *Forward secrecy and backward secrecy of keys*: the compromise of a secret key must not affect the confidentiality of future keys (forward secrecy) and of earlier keys (backward secrecy). In the handover from LTE to UMTS, forward secrecy is specified as the inability of the attacker to derive UMTS keys (CK', IK') when he knows K_{eNB}. Likewise, in the handover from UMTS to LTE, forward secrecy is specified as the inability of the attacker to derive LTE keys (K'_{ASME}, K_{eNB}) when he knows CK and IK. In both X2 and S1 LTE to LTE handovers, forward secrecy is specified as the inability of the attacker to derive the $K^{\star}_{eNB}$ key used in the target eNB when he knows the K_{eNB} used in the source eNB. Backward secrecy is defined as the inability of the attacker to derive K_{eNB} from CK' and IK' in the first case, to derive CK and IK from K_{eNB} in the second case, and to derive K_{eNB} from $K^{\star}_{eNB}$ in the LTE S1 and X2 cases. ProVerif provides a dedicated feature (the `phase` instruction) for checking forward and backward secrecy. The following lines show how forward secrecy is verified in the LTE to UMTS handover [2]:

```
1   ...
2   (* verify forward secrecy *)
3   query attacker(payloadUMTS) phase 1.
4   ...
5   let processMS(uecaps:bset) =
6     (* complete handover procedure *)
```

[2] these lines are not displayed in Figure 5.3 for simplicity

```
7     ...
8     phase 1;
9     out(pubChannel,(kenb_ue));
10    0.
11   ...
```

The `phase` instruction in the `processMS` process breaks the protocol into two phases: phase 0 (the default phase) contains all the instructions and communications that are performed before reaching the instruction `phase 1`. When the latter instruction is reached (i.e. the handover has completed successfully) a new phase (phase 1) begins. In phase 1, only the statements defined after the `phase 1` instruction are executed (in this case, the K_{eNB} key is disclosed), but the adversary keeps all the knowledge acquired during the previous phase (e.g. all the messages exchanged), and integrates it with new terms, if possible (the K_{eNB} key in the example). Similarly, the queries that specify a `phase` n condition are evaluated only after the beginning of phase n. In this excerpt of code, ProVerif evaluates the query when the attacker knows K_{eNB}.

- *Immunity to off-line guessing attacks*: a term is a weak-secret if it is vulnerable to brute-force off-line guessing, and the attacker has the ability to verify if a guessed value is indeed the weak-secret without further interaction after an execution of the protocol. In the handover models, the payloads are data that could be guessed, so it is specified that they must not be weak-secrets. The query `weaksecret`, available in ProVerif to specify that a term must not be a weak-secret, i.e. that the adversary must not be able to distinguish a correct guess of the secret term from an incorrect guess, is used to specify that the payloads must not be weak secrets (lines 16, 17).

- *Authentication*: in the LTE to UMTS and UMTS to LTE handover models, the following authentication properties between the MS and the SN (eNB and RNC) are specified : (i) the MS is authenticated to the source network, (ii) the MS is authenticated to the target network (if the handover procedure has completed successfully), (iii) each time the MS successfully concludes a handover, then the MME previously derived the same keys (K'_{ASME} or CK'/IK'). In the LTE to LTE handover models (both X2 and S1), two authentication queries similar to the first two ones of the LTE to UMTS and UMTS to LTE handovers have been defined: (i) the MS is authenticated to the source eNB, and (ii) the MS is authenticated to the target eNB (if the handover procedure has completed successfully). The third query about the identity of derived keys is useless in this case, because no new key is derived, but the K_{ASME}, K_{NASenc} and K_{NASint} keys, shared between MS and MME, do not change during the handover. Authentication properties are specified as correspondence queries in ProVerif (lines 6 - 12, 29). For example, the

authentication requirement expressed as `inj-event(endMS_ENB(x1,x2))` ⇒ `inj-event(begMS_ENB(x1,x2))` means that each time the event `endMS_ENB (x1,x2)` in the eNB process occurs, the MS process has previously started a session of the protocol (i.e. event `begMS_ENB(x1,x2)` has occurred).

5.5 Verification results

As already explained, all handover types have been analyzed considering both the case that the eNB-MME link includes IPsec protection, and the case that it does not. This produces two different models for each handover type: the two models differ only in the definition of the eNB-MME channel (private in the first case, public in the latter case).

It is worth noting that each property has been verified independently. This is necessary not only for limiting the complexity of the analysis, but also because different properties require different assumptions. For example, when verifying backward/forward secrecy, some keys are intentionally disclosed to the attacker, while the same must not happen when verifying other properties.

5.5.1 LTE to UMTS

Table 5.1 resumes the results of the formal analysis of the LTE to UMTS handover model.

The second column of Table 5.1 contains the results of the analysis when the channel between eNB and MME is private, i.e. the adversary has no access to it. These results confirm that all the expected properties hold: all keys (K_{ASME}, K_{eNB} and derived) remain secret; forward and backward secrecy are valid; the payloads are conditionally secret and are not weak-secrets, and authentication properties hold.

The third column of Table 5.1 refers to the case of a public eNB-MME channel (the adversary can spoof, delete and transmit new messages over the channel). In this scenario, the attacker can know a subset of the LTE keys: K_{eNB} and the derived keys K_{RRCenc}, K_{RRCint} and K_{UPenc}. However, K_{ASME} and the UMTS keys (CK'/IK') are kept secret. The disclosure of K_{eNB} makes the LTE payload not secret (the attacker can derive the ciphering key K_{UPenc}), which also invalids the immunity to guessing attacks on the LTE payload. Instead, the secrecy of the UMTS payload is preserved, because CK remains secret, as well as the immunity to guessing attacks on the UMTS payload. In this scenario, backward secrecy is not valid: the attacker directly knows K_{eNB}. Instead, forward secrecy is kept: the attacker never knows K_{ASME}, so he has no way to derive CK' and IK'. Finally, the authentication between MS and eNB does not hold: an attacker can force a handover of the MS from LTE to UMTS. In fact, the attacker, knowing the IMSI and having access to the eNB-MME channel, can initiate an arbitrary handover by sending a forged HANDOVER REQUIRED

message to the MME. The MS cannot recognize the attacker because the handover procedure continues as in a regular handover, and receives a genuine HANDOVER COMMAND message from the network. The attacker never knows K_{ASME}: if the handover completes in the MS, then the MME must have previously derived, in a corresponding session, the CK′ and IK′ keys from K_{ASME}, so MME and MS are correctly authenticated during the handover. Similarly, the attacker has no access to the 3G serving network and, from the previous properties, to the CK′ and IK′ keys: the attacker cannot alter communications between RNC and MS and, when the handover procedure completes, the MS and the UMTS SN are authenticated.

	LTE to UMTS	
eNB-MME channel	private	public
Secrecy of keys	true	false for K_{eNB} and keys derived from K_{eNB}
Conditional secrecy of LTE payload	true	false
Conditional secrecy of UMTS payload	true	true
Forward secrecy	true	true
Backward secrecy	true	false
Immunity to off-line guessing attacks	true	false for payloadLTE, true for payloadUMTS
Auth. MS-eNB	true	false
Auth. MS-MME	true	true
Auth. MS-RNC	true	true

Table 5.1. Analysis results: LTE to UMTS handover

5.5.2 UMTS to LTE

The same considerations made for the two previous scenarios are also applicable to the other handover procedure, from UMTS to LTE (second and third column in Table 5.2), with only some differences. The only results that differ are the ones about forward and backward secrecy. In this handover scenario, forward secrecy does not hold because if the attacker knows CK and IK, he can decrypt all the messages between MS and the UMTS network. In this way, the adversary can read the nonce, transmitted from the RNC to the MS, that is used by MME and MS, along with CK and IK, to derive the K'_{ASME} key, and subsequently all the LTE keys. Instead, backward secrecy holds: an attacker who knows K_{eNB} cannot derive the previous CK and IK keys.

The results about authentication are the same, albeit their explanation is different.

Lack of authentication between MS and eNB, in the last scenario, makes the adversary able to alter all subsequent Access Stratum and User Plane communications between MS and eNB. However, the attacker cannot read and modify Non Access Stratum messages between MS and MME. For this reason MS-MME authentication remains valid: if the handover completes in the MS, then the MME ran a session where the K_{ASME} key was derived, so MME and MS are authenticated during the handover. Finally, before starting the handover, MS-RNC are authenticated, as confirmed by the last query, because the attacker has no access to the UMTS network.

	UMTS to LTE	
eNB-MME channel	private	public
Secrecy of keys	true	false for K_{eNB} and keys derived from K_{eNB}
Conditional secrecy of LTE payload	true	false
Conditional secrecy of UMTS payload	true	true
Forward secrecy	false	false
Backward secrecy	true	true
Immunity to off-line guessing attacks	true	false for payloadLTE, true for payloadUMTS
Auth. MS-eNB	true	false
Auth. MS-MME	true	true
Auth. MS-RNC	true	true

Table 5.2. Analysis results: UMTS to LTE handover

5.5.3 LTE X2

Table 5.3 resumes the results of the formal analysis of the LTE X2 handover model.

In this handover scenario, for the three channels has been considered the possibility that each channel may be insecure. Thus, a total of eight combinations are possible, when channels are alternatively considered as private or public channels. In certain cases, ProVerif is not able to verify all the properties ("unres", i.e. unresolved, cells in Table 5.3).

In the X2 handover, forward secrecy never holds, as already known from the specifications [53] (TS 33.401).

The columns of Table 5.3 confirm that the security properties of the current handover procedure are not influenced by the protection on the TeNB-MME channel: this can be explained because the only key that is transmitted on that channel is the Next Hop Key 2, which will be eventually used in the next handover. However, the

next handover may be compromised if the attacker has the Next Hop Key 2. If this happens, during the following handover the security properties will not hold.

The fourth and fifth columns consider the case when the the SeNB-TeNB channel is protected while the SeNB-MME channel lacks protection. In this scenario, the attacker obtains K_{eNB} from the second message, and can derive all the subsequent keys. Moreover, if the TeNB-MME channel is also unprotected (fifth column), the attacker can read the Next Hop Key 2 sent by the MME. Conditional secrecy of payloads is not true, because the ciphering keys are disclosed (ProVerif is not able to resolve the query about payload 2). This implies that the payloads are also reported as weak-secrets, because the attacker knows the exact values from the previous point. Similarly, backward secrecy is not valid because K_{eNB} is directly known by the attacker. Finally, authentication does not hold: the attacker obtains all the keys needed in the handover procedure, thus he can act as fake SeNB and TeNB. Unfortunately, ProVerif cannot resolve the query about the authentication between MS and SeNB, i.e. it cannot complete this verification. However, it can be argued that if the attacker has K_{eNB}, he can replicate the behaviour of the SeNB, thus invalidating this authentication.

The sixth to ninth columns of Table 5.3 consider the case when the channel between SeNB and TeNB (the X2 interface) is not protected. In this scenario it is clear that the attacker always knows $K^{\star}_{eNB}$. The direct effect is that the authentication between MS and TeNB never holds: in fact the attacker may operate as fake TeNB because all the keys are derived from $K^{\star}_{eNB}$. In particular, the attacker can arbitrarily force a handover execution, by sending a forged HANDOVER REQUEST message to the TeNB. Moreover, Table 5.3 shows that the protection of the TeNB-MME channel does not influence the security properties apart from the fact that the Next Hop Key 2 is disclosed if the TeNB-MME and SeNB-TeNB channels are public. When the SeNB-MME channel is private (the attacker does not know K_{eNB} and K_{UPenc}), the conditional secrecy of payload 1 (sent before the handover begins) holds, while payload 2 (sent after the handover completion) is always known by the attacker (because it is ciphered with the K_{UPenc} derived from $K^{\star}_{eNB}$), thus the conditional secrecy of payload 2 is false (ProVerif is not able to resolve the queries when the the SeNB-MME channel is public). Similarly, backward secrecy holds only if the SeNB-MME channel is private. Otherwise, the attacker can obtain K_{eNB} and invalidate the property. Moreover, payload 1 cannot be guessed if the SeNB-MME channel is private: the attacker cannot derive K_{UPenc} because he does not know K_{eNB}. Finally, authentication between MS and SeNB holds only if the SeNB-MME channel is protected. If it is not, the attacker can behave as a fake SeNB (ProVerif is not able to resolve this query).

	LTE X2							
SeNB-TeNB channel	private				public			
SeNB-MME channel	private		public		private		public	
TeNB-MME channel	private	public	private	public	private	public	private	public
Secrecy of keys	true	true	false for K_{eNB} and derived	false for K_{eNB} and derived and NH2 key	false for $K^{\star}_{eNB}$ and derived	false for $K^{\star}_{eNB}$ and derived	false (except K_{ASME} and NH1 key)	false (except K_{ASME}, NH1 and NH2 keys)
Conditional secrecy of LTE 1 payload	true	true	false	false	true	true	false	false
Conditional secrecy of LTE 2 payload	true	true	unres	unres	false	false	unres	unres
Forward secrecy	false	false	false	false	false	false	false	false
Backward secrecy	true	true	false	false	true	true	false	false
Immunity to off-line guessing attacks	true	true	false	false	true for payload 1, false for payload 2	true for payload 1, false for payload 2	false	false
Auth. MS-SeNB	true	true	unres	unres	true	true	unres	unres
Auth. MS-TeNB	true	true	false	false	false	false	false	false

Notes:
unres = unresolved, i.e. ProVerif cannot resolve the query
NH1 = Next Hop 1 key
NH2 = Next Hop 2 key

Table 5.3. Analysis results: LTE X2 handover

5.5.4 LTE S1

Table 5.4 resumes the results of the formal analysis of the LTE S1 handover model.

In this handover scenario, for the SeNB-MME and TeNB-MME channels, both the case of protected channel and the case of unprotected channel have been considered, for a total of four different scenarios (note that in this kind of handover there is no SeNB-TeNB channel, see Section 5.4.2).

The second column of Table 5.4 considers the case when both channels are private: all the security properties are verified. Conversely, if both channels are modeled as public channels, none of the properties is verified (ProVerif is not even able to resolve all the queries), as reported in the fifth column of Table 5.4.

If the SeNB-MME channel is private and the TeNB-MME channel is public (third column of Table 5.4), the attacker may obtain all the keys used in the TeNB, because all the keys are derived from the Next Hop 2 and the Target Cell ID (which is public). The attacker does not know the keys used in the SeNB, which implies that the conditional secrecy of payload 1 holds. ProVerif is not able to resolve the query about payload 2. However, payload 2 is known by the attacker because he knows all

the keys used in the TeNB. The fact that the attacker has all the keys derived in the TeNB also falsifies the queries about forward secrecy (because the attacker may derive $K^{\star}_{eNB}$), and about the MS-TeNB authentication (the attacker has all the keys to act as TeNB). Backward secrecy is verified, which can be explained because the attacker has no way to obtain the initial K_{eNB}. Finally, payload 1 cannot be guessed offline, but payload 2 is known by the attacker because it is received by the TeNB, and the attacker has the keys used in th TeNB.

	LTE S1			
SeNB-MME channel	private		public	
TeNB-MME channel	private	public	private	public
Secrecy of keys	true	unres for HH2 and TeNB keys	unres	unres
Conditional secrecy of LTE 1 payload	true	true	unres	unres
Conditional secrecy of LTE 2 payload	true	unres	true	unres
Forward secrecy	true	false	true	false
Backward secrecy	true	true	false	false
Immunity to off-line guessing attacks	true	true for payload 1, false for payload 2	true for payload 2, false for payload 1	false
Auth. MS-SeNB	true	true	unres	unres
Auth. MS-TeNB	true	false	true	false

Table 5.4. Analysis results: LTE S1 handover

The last scenario, which results are reported in the fourth column of Table 5.4, considers the case when the SeNB-MME channel is public and the TeNB-MME channel is private. ProVerif is not able to resolve the queries about the secrecy of the keys. However, from the model it is clear that the attacker knows K_{eNB} (from the second message sent by the MME to the SeNB), and is able to derive all the keys used by the SeNB. Thus, the attacker can obtain payload 1, which falsifies its conditional secrecy and off-line guessing resistance. Finally, the attacker may act as SeNB: the authentication between MS and SeNB is not verified by ProVerif, and the attacker can force a handover execution, by sending a forged HANDOVER REQUIRED message to the MME (this is also possible when both channels are public, fifth column). Since the TeNB-MME channel is private, the attacker does not know the keys used in the TeNB. Payload 2 remains conditionally secret and resistant to off-line guessing. Similarly, forward secrecy holds, which can be explained because $K^{\star}_{eNB}$, derived from TeNB, is not known by the attacker, while the backward

secrecy query is falsified because the attacker directly knows K_{eNB} from the second message (sent by the MME to the SeNB). Finally, the authentication between MS and TeNB holds, which can be explained because the attacker is not able to obtain the keys used in the TeNB.

5.6 Related work

In [7] preliminary results of the formal analysis of the handover procedures between LTE and UMTS were presented. This thesis extends the results by also considering emergency calls and provides full details about the formal models used for verification, along with an explanation of design choices. Moreover, the analysis of the handover procedures between LTE and UMTS has been completed with a thorough formal analysis of the LTE X2 and LTE S1 procedures.

Ben Henda and Norrman [54] recently used ProVerif to analyze the LTE procedures related to session management (used to establish security algorithms between the mobile device and the network) and mobility (handover between two LTE cells). The procedures analyzed are: Network Access Stratum (NAS) security control procedure, i.e. security algorithm negotiation between MS and MME, NAS Service Request Procedure (security algorithm negotiation between MS and eNodeB), X2 handover, and S1 handover. The reported results show that secrecy and agreement properties hold as expected. However, differently from the work described in this part of the thesis, the analysis proposed in [54] does not consider the possibility that data encryption may be disabled and that some channels may lack IPSec protection as allowed by the standard [53] (TS 33.401). Moreover, Ben Henda and Norrman [54] do not consider the possibility of having emergency calls. Finally, the work described here checks also weak-secrecy, i.e. prove that the adversary cannot distinguish a correct guess of a secret term from an incorrect guess.

The research community mainly focused on analyzing the Authentication and Key Agreement (AKA) procedure and on proposing improvements in that procedure [57], [58], [59] and [60]. LTE and UMTS authentication procedures are very similar, and only computation of keys and used algorithms differ. The UMTS AKA was formally analyzed using BAN logic in TS 33.902 [53] and, due to the similarity of the procedures, all analysis results carry over to LTE AKA.

Arapinis et al. [56] used ProVerif to analyze privacy aspects of UMTS. However, the paging procedure analyzed is the same in LTE and UMTS technologies, so the results should be valid for both networks.

Qachri et al. [61] propose and analyze a system for handovers between different wireless network technologies (e.g. 3G, 4G, WiFi, WiMax). The proposed system has been formally verified with ProVerif. However, the paper does not provide an analysis of the LTE network defined by the 3GPP standards.

Part III

Conclusion

Chapter 6

Conclusion

Formal verification techniques can be applied in various fields, e.g. software applications, hardware devices, or mixed software/hardware systems. The formal verification procedures exploit mathematical models and logical reasoning in order to establish if the system under analysis satisfies certain properties. However, building a formal model is not a simple task: they must be proper a abstraction of the real system, without being too complex (because they would be unfeasible to analyze) and, at the same time, not too simple (in order to obtain significant results). Throughout the years, different methodologies, tools and frameworks have been proposed in order to reduce the complexity of formal verification processes, and enable inexperienced developers to use it, by automating some phases or all of the verification process.

Part I of this thesis presented Spi2JavaGUI and JavaSPI: two frameworks that facilitate the use of formal verification techniques in the verification of distributed applications. In particular, these solutions lower the adoption barriers for developers that are not formal verification experts, and reduce the resources necessary to complete the verification than previous existing solutions. In particular, Spi2JavaGUI combines a graphical editor (implemented as a plugin for the Eclipse platform), formal verification (with symbolic model), and the generation of Java code that implements the protocol. Instead, JavaSPI enables the verification of application dependent security properties, in addition to generic security properties (i.e. secrecy and integrity of data, authentication of parties). The former class contains properties that heavily depends on the application under examination. For example, the value of a variable must always be greater than zero for all possible execution paths that can be followed when the different processes (that compose the distributed application) are running.

Part II of the thesis presented the formal analysis of handover procedures, defined in the Long Term Evolution (LTE) standard for mobile communication networks, that were not previously analyzed in literature. The ProVerif tool has been used to perform the formal analysis. Finally, a detailed description of the methodology used to translate the LTE standards to ProVerif models has been given.

The next sections resume the results obtained and possible future evolutions and

improvements.

6.1 Spi2JavaGUI

Spi2JavaGUI is a domain-specific approach for visual MDD of security protocols. This approach combines visual editing, which is more intuitive than existing textual formal languages, with a rigorous formal approach to model verification and code generation.

The visual model hides the complexity of textual formal notation at different levels of abstraction. At the same time, the soundness of the Spi2Java framework upon which Spi2JavaGUI builds provides the possibility to generate code with high confidence of correctness.

It has been shown how a security protocol can be modeled with Spi2JavaGUI, how to perform formal analysis and how to generate the final code that implements the protocol, following the MDD paradigm. In conclusion, the approach described here makes it possible to easily use the of Spi2Java framework without the need to know the syntax of spi calculus language.

Spi2JavaGUI is implemented as a set of plug-ins for the Eclipse platform, which is widely used in the software modeling and development community. The Eclipse platform provides advanced editing facilities and facilitates deployment and distribution.

Currently, Spi2JavaGUI covers almost all functionality of Spi2Java framework, then all security considerations about Spi2Java can be applied also to Spi2JavaGUI.

Compared to the related works presented in Section 3.3, Spi2JavaGUI can be considered a good step ahead, because none of the related techniques provides at the same time intuitive visual modeling, formal analysis and sound generation of interoperable code for the whole class of security protocols. Also, the Spi2JavaGUI visual models differ substantially from previous proposals. Compared to UML-based approaches, which use many views and diagrams, Spi2JavaGUI provides a single comprehensive view tailored on the specific needs of security protocol modeling. Compared to other approaches that provide linkage with formal models, Spi2JavaGUI is more oriented to software developers because the data-flow-oriented models used by Spi2JavaGUI are easy to understand and provide enough implementation detail to generate interoperable implementation code automatically.

Spi2JavaGUI visual models substantially differ from previous proposals. While UML-based approaches use many views and diagrams, Spi2JavaGUI provides a single comprehensive view, tailored to the specific needs of security protocol modeling. Furthermore, with respect to other approaches providing linkage with formal models that are oriented to experts in formal methods and security, Spi2JavaGUI targets software developers by means of using data-flow-oriented models that are similar to other models well-known by software developers, and thus of easy understanding for

software developers. These models also provide enough implementation details to enable automatic code generation.

The proposed Spi2JavaGUI formalism fulfills the main key characteristics for a model-driven approach, as described by Selic [1]. *Abstraction* is enforced by view collapsing and hierarchical models. *Understandability* is achieved by visualizing the protocol data path of each protocol session in MSC-like style. In addition, the Spi2JavaGUI approach clearly fulfills *accuracy*, i.e. true-to-life representation and *predictiveness*, i.e. possibility to predict interesting but non-obvious properties, because of its ability to discover attacks on the modeled protocol. Even if not verified experimentally, it is believable that the Spi2JavaGUI approach also fulfills *inexpensiveness*, i.e. the fact that models are significantly cheaper to construct and analyze than the modeled system, because Spi2JavaGUI models are relatively simple and let the user focus on the protocol logic alone before being involved in other implementation details.

Empirical evaluations of Spi2JavaGUI, which may complete its assessment, and are left as future work.

The implementation of the framework is still at an early stage. Graphical appeal, flexibility and usability of the interface can be further improved.

The integration with the automatic protocol verifier ProVerif can be further improved (currently the result of analysis is only textual), by reporting results on the graphical interface, visually simulating the attacks on the protocol, as found by the tool.

The Spi2Java framework can also be improved in some of its features. For example, one of the current limitations of this framework is that it covers only the protocol logic while it leaves the encoding and decoding functions to be written manually. Automating the generation of code for this software layer is a possible research aim.

An idea of how it can be done is to describe abstract data types with languages like ASN.1 or NetPDL, then, using already existing tools (or by developing a new tool), the encoding libraries may be automatically generated. With this functionality, the user who wants to customize the encoding layer of a protocol is no more required to handle the programming language in details.

Another possible extension to Spi2JavaGUI consists in widening its scope from security protocols to distributed cloud applications and mobile environments.

6.2 Automated formal verification of application-specific security properties

Chapter 4 described how a distributed application with application-specific security requirements can be developed using a model-driven approach that finally yields

a formally verified Java implementation. The formal verification of the security properties takes into account active attackers and is entirely automated. The most critical part of the code, i.e. the implementation of the security protocol, is generated automatically from an abstract model with the guarantee of security property preservation. Moreover, the model is written in Java, instead of using domain-specific formal languages. The adoption of a compositional verification approach splits verification into two separate simpler tasks, which potentially leads to the possibility to handle larger applications.

Currently, no other approach was previously proposed with all these features together. Compared to the approach presented in [38], which developed the same case study, the approach described here has the advantage of being fully automated. Even if model checking does not allow to get a result that holds for any number of users and terminals, the result gives anyway good security assurance and can be obtained using only automated tools and without requiring excessive expertise.

The results obtained are encouraging because they confirm that it is possible to develop distributed applications with formally verified application-specific security properties using only automated tools.

One drawback is the high quantity of resources that the model checking with JPF requires, in terms of memory and time. This is partially due to the kind of verification that interprets the bytecode of the real Java application. Using other verification tools for Java may improve the performance. Future works will address the verification of generic security properties in the final application code, for example guarantee that the value of a field added manually remains confidential in the final application.

6.3 LTE and UMTS handover procedures

LTE is the most recent standard in communication systems developed by 3GPP. Part II of this thesis presented a thorough formal security analysis of handover procedures activated when a mobile device moves between LTE and UMTS networks or between LTE nodes. The tool used to formalize models and to verify procedures is ProVerif, which uses symbolic models based on perfect cryptography assumptions. The results about UMTS-LTE handovers already presented in [7] have been extended with the analysis of new verification scenarios in the presence of emergency calls (in order to check if an attacker can exploit emergency sessions to break the security of the network) and by giving full details about the formal models used for verification and the design choices adopted in their definition. The results about LTE to LTE handovers (X2 and S1) that were available in the literature have been completed with new results that consider new kinds of properties and new assumptions not previously considered in the literature. In particular, the results already presented by Ben Henda and Norrman [54], regrading authentication and secrecy in LTE X2 and

S1 handovers, have been confirmed by this thesis. For all the considered handover procedures, secrecy of ciphering and integrity keys, conditional secrecy of payloads, forward and backward secrecy of keys, immunity to guessing attacks on payloads and authentication between network components have been analyzed.

3GPP specifies that mobile operators can decide to omit IPsec protection on eNB-MME and eNB-eNB channels, if the interfaces are trusted. However, a definition of "trusted" is not given by 3GPP specifications, but it is left to the mobile operators' discretion. As currently several operators do not protect the eNB-MME and eNB-eNB channels, the analysis was conducted by considering both the cases of protected and unprotected eNB-MME and eNB-eNB channels.

Results confirm that, under the assumptions made, almost all the properties that have been considered hold when eNB-MME and eNB-eNB channels are protected in all the four handover procedures. The only property that does not hold is forward secrecy (as defined in Section 5.4.2) in the UMTS to LTE and the X2 handovers. Moreover, it is possible to confirm that the emergency sessions do not disclose to the attackers data that can be used to break network security during handover procedures.

In the case of unprotected eNB-MME or eNB-eNB channels, results show which properties are broken and which remain valid under the assumptions made. When having access to the eNB-MME channels, an attacker can force a handover from LTE to UMTS, or control the Access Stratum and User Plane communications after a handover from UMTS to LTE. However, the main LTE key (K_{ASME}) and the UMTS keys (CK'/IK') are kept secret.

In the LTE to LTE procedures a greater number of combinations are possible, because the channels that may be considered insecure are 2 (S1 handover), or 3 (X2 handover). In both the handover cases, the attacker can alter sections, or the entire handover process, depending on which channels he controls.

Finally, results highlight that the handover procedure from UMTS to LTE does not provide forward secrecy of the keys, with respect to the definition given in Section 5.4.2. Similarly, the X2 handover never guarantees forward secrecy, but this is a precise 3GPP design choice in order to obtain a very fast handover procedure, which is particularly useful for fast-moving users and devices.

A total of 16 ProVerif models have been analyzed. All the handover procedure were verified considering the possibility that the attacker can control the channel between eNB and MME and between eNB and eNB.

References

[1] Bran Selic. The pragmatics of model-driven development. *IEEE Softw.*, 20: 19–25, September 2003. ISSN 0740-7459.

[2] Piergiuseppe Bettassa Copet, Alfredo Pironti, Davide Pozza, Riccardo Sisto, and Pietro Vivoli. Visual model-driven design, verification and implementation of security protocols. In *HASE*, pages 62–65, 2012.

[3] M. Avalle, A. Pironti, R. Sisto, and D. Pozza. The Java SPI framework for security protocol implementation. In *Availability, Reliability and Security (ARES), 2011 Sixth International Conference on*, pages 746 –751, aug. 2011.

[4] Matteo Avalle, Alfredo Pironti, Davide Pozza, and Riccardo Sisto. JavaSPI: A framework for security protocol implementation. *International Journal of Secure Software Engineering*, 2(4):34–48, 2011.

[5] Piergiuseppe Bettassa Copet and Riccardo Sisto. Automated formal verification of application-specific security properties. In *Engineering Secure Software and Systems*, volume 8364 of *Lecture Notes in Computer Science*, pages 45–59. Springer International Publishing, 2014. ISBN 978-3-319-04896-3.

[6] Matteo Avalle. *New Techniques to Improve Network Security.* PhD thesis, Politecnico di Torino, 2014.

[7] Piergiuseppe Bettassa Copet, Guido Marchetto, Riccardo Sisto, and Luciana Costa. Formal verification of LTE-UMTS handover procedures. In *IEEE Symposium on Computers and Communication (ISCC)*, pages 738–744, July 2015.

[8] B. Moeller. Security of CBC ciphersuites in SSL/TLS: Problems and countermeasures. http://www.openssl.org/ bodo/tls-cbc.txt, 2004.

[9] Marsh Ray and S Dispensa. Authentication gap in TLS renegotiation, 2009.

[10] S. Turner and T. Polk. Prohibiting secure sockets layer (SSL) version 2.0. RFC 6176, 2011.

[11] Apple goto fail bug, 2014. URL `http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2014-1266`. CVE-2014-1266.

[12] GnuTLS certificate verification issue, 2014. URL `http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2014-0092`. CVE-2014-0092.

[13] Heartbleed bug, 2014. URL `http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2014-0160`. CVE-2014-0160.

[14] Matteo Avalle, Alfredo Pironti, and Riccardo Sisto. Formal verification of security protocol implementations: a survey. *Formal Aspects of Computing*, 26 (1):99–123, 2012. ISSN 1433-299X.

[15] Bruno Blanchet. An efficient cryptographic protocol verifier based on Prolog rules. In *14th IEEE workshop on Computer Security Foundations*, pages 82–96, 2001.

[16] Bruno Blanchet. Automatic verification of correspondences for security protocols. *Journal of Computer Security*, 17(4):363–434, 2009.

[17] Danny Dolev and Andrew Chi-Chih Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, 29(2):198–207, 1983.

[18] Willem Visser, Klaus Havelund, Guillaume Brat, Seungjoon Park, and Flavio Lerda. Model checking programs. *Automated Software Engg.*, 10(2):203–232, 2003. ISSN 0928-8910.

[19] D. Pozza, R. Sisto, and L. Durante. Spi2Java: automatic cryptographic protocol Java code generation from spi calculus. In *Advanced Information Networking and Applications, 2004. 18th International Conference on*, volume 1, pages 400 – 405 Vol.1, 2004.

[20] Alfredo Pironti and Riccardo Sisto. An experiment in interoperable cryptographic protocol implementation using automatic code generation. In *IEEE Symposium on Computers and Communications*, pages 839–844, 2007.

[21] Martín Abadi and Andrew D. Gordon. A calculus for cryptographic protocols: the spi calculus. In *Proceedings of the 4th ACM conference on Computer and communications security*, pages 36–47, 1997. ISBN 0-89791-912-2.

[22] Alfredo Pironti, Davide Pozza, and Riccardo Sisto. Formally-based semi-automatic implementation of an open security protocol. *Journal of Systems and Software*, 85:835–849, 2012.

[23] A. Pironti and R Sisto. Provably correct Java implementations of Spi Calculus security protocols specifications. *Computers & Security*, 29:302–314, 2010.

[24] Gavin Lowe. A hierarchy of authentication specifications. In *Proceedings of the 10th IEEE workshop on Computer Security Foundations*, pages 31–43, 1997. ISBN 0-8186-7990-5.

[25] J. McDermott. Visual security protocol modeling. In *Proceedings of the 2005 workshop on New security paradigms*, pages 97–109, 2005. ISBN 1-59593-317-4.

[26] C. A. R. Hoare. *Communicating sequential processes.* 1985. ISBN 0-13-153271-5.

[27] Jane Hillston. *A compositional approach to performance modelling.* 1996. ISBN 0-521-57189-8.

[28] Elton Saul and Andrew Hutchison. Using GYPSIE, GYNGER and Visual GNY to analyze cryptographic protocols in Spear II. In *Advances in Information Security Management & Small Systems Security*, volume 72, pages 73–85. 2001. ISBN 978-0-7923-7506-7.

[29] Michael Burrows, Martin Abadi, and Roger Needham. A logic of authentication. *ACM Trans. Comput. Syst.*, 8(1):18–36, February 1990. ISSN 0734-2071.

[30] Pete Epstein and Ravi Sandhu. Towards a UML based approach to role engineering. In *Proceedings of the fourth ACM workshop on Role-based access control*, pages 135–143, 1999. ISBN 1-58113-180-1.

[31] David Basin, Jürgen Doser, and Torsten Lodderstedt. Model driven security for process-oriented systems. In *Proceedings of the eighth ACM symposium on Access control models and technologies*, pages 100–109, 2003. ISBN 1-58113-681-1.

[32] Aisha Bushager and Mark Zwolinski. Evaluating system security using transaction level modelling. *Electronics and Energetics*, 27(1):137–151, March 2014.

[33] *IEEE Standard System C Language Reference Manual.* IEEE Std 1666, 2005.

[34] Jan Jürjens. UMLsec: Extending UML for Secure Systems Development. In *Proceedings of the 5th International Conference on The Unified Modeling Language*, pages 412–425, 2002. ISBN 3-540-44254-5.

[35] Jan Jürjens. Developing high-assurance secure systems with UML: a smartcard-based purchase protocol. In *Proceedings of the Eighth IEEE international conference on High assurance systems engineering*, pages 231–240, 2004. ISBN 0-7695-2094-4.

[36] Jan Jürjens. Model-based security testing using UMLsec. *Electron. Notes Theor. Comput. Sci.*, 220(1):93–104, December 2008. ISSN 1571-0661.

[37] N. Moebius, K. Stenzel, H. Grandy, and W. Reif. SecureMDD: A model-driven development method for secure smart card applications. In *Availability, Reliability and Security, 2009. International Conference on*, pages 841 –846, march 2009.

[38] Nina Moebius, Kurt Stenzel, and Wolfgang Reif. Formal verification of application-specific security properties in a model-driven approach. In *Engineering Secure Software and Systems*, volume 5965 of *Lecture Notes in Computer Science*, pages 166–181, 2010. ISBN 978-3-642-11746-6.

[39] Marian Borek, Kuzman Katkalov, Nina Moebius, Wolfgang Reif, Gerhard Schellhorn, and Kurt Stenzel. Integrating a model-driven approach and formal verification for the development of secure service applications. In Bernhard Thalheim, Klaus-Dieter Schewe, Andreas Prinz, and Bruno Buchberger, editors, *Correct Software in Web Applications and Web Services*, Texts & Monographs in Symbolic Computation, pages 45–81. Springer International Publishing, 2015. ISBN 978-3-319-17111-1.

[40] S. Smith, A. Beaulieu, and W.G. Phillips. Modeling and verifying security protocols using UML 2. In *Systems Conference (SysCon), 2011 IEEE International*, pages 72 –79, 2011.

[41] Reema Patel, Bhavesh Borisaniya, Avi Patel, Dhiren Patel, Muttukrishnan Rajarajan, and Andrea Zisman. Comparative analysis of formal model checking tools for security protocol verification. In Natarajan Meghanathan, Selma Boumerdassi, Nabendu Chaki, and Dhinaharan Nagamalai, editors, *Recent Trends in Network Security and Applications*, volume 89 of *Communications in Computer and Information Science*, pages 152–163. Springer Berlin Heidelberg, 2010. ISBN 978-3-642-14477-6.

[42] Paolo Modesti. Efficient Java code generation of security protocols specified in AnB/AnBx. In Sjouke Mauw and ChristianDamsgaard Jensen, editors, *Security and Trust Management*, volume 8743 of *Lecture Notes in Computer Science*, pages 204–208. Springer International Publishing, 2014. ISBN 978-3-319-11850-5.

[43] Samir Ouchani and Mourad Debbabi. Specification, verification, and quantification of security in model-based systems. *Computing*, 97(7):691–711, 2015. ISSN 0010-485X.

[44] Phu Hong Nguyen, Max E. Kramer, Jacques Klein, and Yves Le Traon. An extensive systematic review on model-driven development of secure systems. *CoRR*, abs/1505.06557, 2015.

[45] Giampaolo Bella, Fabio Massacci, and Lawrence C. Paulson. Verifying the SET purchase protocols. *J. Autom. Reason.*, 36(1-2):5–37, 2006. ISSN 0168-7433.

[46] Tobias Nipkow, Markus Wenzel, and Lawrence C. Paulson. *Isabelle/HOL: a proof assistant for higher-order logic*. Springer-Verlag, Berlin, Heidelberg, 2002. ISBN 3-540-43376-7.

[47] Marian Borek, Nina Moebius, Kurt Stenzel, and Wolfgang Reif. Model-driven development of secure service applications. In *Proceedings of the 35th Annual IEEE Software Engineering Workshop (SEW)*, pages 62–71. IEEE, 2012.

[48] Marian Borek, Nina Moebius, Kurt Stenzel, and Wolfgang Reif. Model checking of security-critical applications in a model-driven approach. In Robert M. Hierons, Mercedes G. Merayo, and Mario Bravetti, editors, *SEFM*, volume 8137 of *Lecture Notes in Computer Science*, pages 76–90. Springer, 2013. ISBN 978-3-642-40560-0.

[49] Alessandro Armando, Wihem Arsac, Tigran Avanesov, Michele Barletta, Alberto Calvi, Alessandro Cappai, Roberto Carbone, Yannick Chevalier, Luca Compagna, Jorge Cuéllar, Gabriel Erzse, Simone Frau, Marius Minea, Sebastian Mödersheim, David von Oheimb, Giancarlo Pellegrino, Serena Elisa Ponta, Marco Rocchetto, Michaël Rusinowitch, Mohammad Torabi Dashti, Mathieu Turuani, and Luca Viganò. The AVANTSSAR platform for the automated validation of trust and security of service-oriented architectures. In *TACAS*, volume 7214 of *Lecture Notes in Computer Science*, pages 267–282. Springer, 2012.

[50] Linda Ariani Gunawan, Frank Alexander Kraemer, and Peter Herrmann. A tool-supported method for the design and implementation of secure distributed applications. In *Proceedings of the Third international conference on Engineering Secure Software and Systems (ESSoS)*, pages 142–155, Berlin, Heidelberg, 2011. Springer-Verlag.

[51] Linda Ariani Gunawan and Peter Herrmann. Compositional verification of application-level security properties. In *Proceedings of the Fifth international conference on Engineering Secure Software and Systems (ESSoS)*, pages 75–90, Berlin, Heidelberg, 2013. Springer-Verlag.

[52] Maria Vasilevskaya, Linda Ariani Gunawan, Simin Nadjm-Tehrani, and Peter Herrmann. Integrating security mechanisms into embedded systems by domain-specific modelling. *Security and Communication Networks*, 7(12):2815–2832, 2014. ISSN 1939-0122.

[53] 3rd Generation Partnership Project (3GPP). 3GPP specifications. `http://www.3gpp.org/specifications`, December 2015.

[54] Noomene Ben Henda and Karl Norrman. Formal analysis of security procedures in LTE - a feasibility study. In Angelos Stavrou, Herbert Bos, and Georgios

Portokalidis, editors, *Research in Attacks, Intrusions and Defenses*, volume 8688 of *Lecture Notes in Computer Science*, pages 341–361. Springer International Publishing, 2014. ISBN 978-3-319-11378-4.

[55] Patrick Donegan. The security vulnerabilities of LTE: Risks for operators. *Juniper Networks white paper*, 2013.

[56] Myrto Arapinis, Loretta Mancini, Eike Ritter, Mark Ryan, Nico Golde, Kevin Redon, and Ravishankar Borgaonkar. New privacy issues in mobile telephony: Fix and verification. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, pages 205–216, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1651-4.

[57] Chunyu Tang, David A. Naumann, and Susanne Wetzel. Symbolic analysis for security of roaming protocols in mobile networks. In Muttukrishnan Rajarajan, Fred Piper, Haining Wang, and George Kesidis, editors, *Security and Privacy in Communication Networks*, volume 96 of *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering*, pages 480–490. Springer Berlin Heidelberg, 2012. ISBN 978-3-642-31908-2.

[58] Joe-Kai Tsay and StigF. Mjølsnes. A vulnerability in the UMTS and LTE authentication and key agreement protocols. In Igor Kotenko and Victor Skormin, editors, *Computer Network Security*, volume 7531 of *Lecture Notes in Computer Science*, pages 65–76. Springer Berlin Heidelberg, 2012. ISBN 978-3-642-33703-1.

[59] Muxiang Zhang and Yuguang Fang. Security analysis and enhancements of 3GPP authentication and key agreement protocol. *Wireless Communications, IEEE Transactions on*, 4(2):734–742, March 2005. ISSN 1536-1276.

[60] Jiexiang Fang and Rui Jiang. An analysis and improvement of 3GPP SAE AKA protocol based on strand space model. In *Network Infrastructure and Digital Content, 2010 2nd IEEE International Conference on*, pages 789–793, Sept 2010.

[61] Naïm Qachri, Olivier Markowitch, and Jean-Michel Dricot. A formally verified protocol for secure vertical handovers in 4G heterogeneous networks. *International Journal of Security and Its Applications*, 2013.